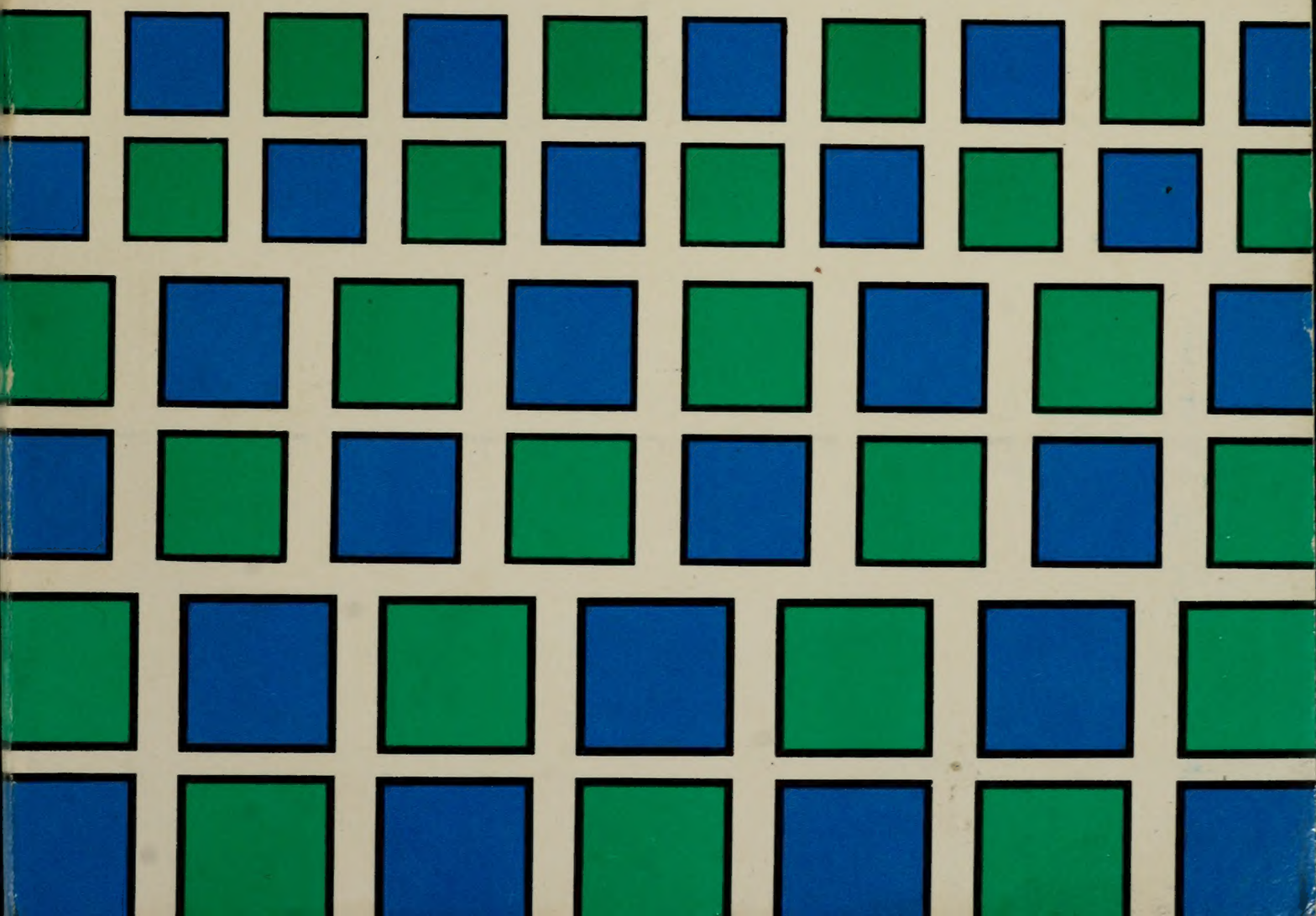
A REWARD BOOK

$7.95

How to Debug Your Personal Computer

Jim Huffman/Robert C. Bruce

Do-it-yourself guide to debugging. Learn how to recognize, track, and eliminate problems in your program or system—the easy way!



How to Debug Your Personal Computer

How to Debug Your Personal Computer

Robert Bruce Jim Huffman

RESTON PUBLISHING, INC., Reston, Virginia
A Prentice-Hall Company

Library of Congress Cataloging in Publication Data

Bruce, Robert C
How to debug your personal computer.

Includes index.
1. Debugging in computer science.
I. Huffman, Jim. II. Title.
QA76.6. B775 001.64 80-18091
ISBN: 0-8359-2924-8

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia 22090

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

Contents

Introduction

A program that should but won't run on your computer is a source of building frustration, especially if you are just beginning to write your own or just paid good money for some new software! Short of starting again at Square One or sending the program back, what can you do about it? *Debug!*

This book will show you how you can tell when there is a bug in a program or your system, how to trakc it down to tis source, and how to get rid of it or get around it. If you have some knowledge of BASIC, you can put the techniques described in this book to work. All other information is included.

You would be absolutely correct in assuming that this is a book on the basics of debugging. In this respect, nothing is presumed. What we attempt to do here is show you how—step-by-step—to recognize and eliminate bugs when you're writing a program and how to spot them during the shakedown period. Also there are techniques for finding bugs in a program you didn't write, easy-to-follow guides to help you determine where in the program a bug is hiding and how to either eliminate it or work around it.

Of course, it is always easier to see bugs on a flowchart. In Chapter 1 you'll earn how to recognize them and how to construct a flowchart for and existing program. If the bug can't be eradicated by a change in the flowchart, the miost effective and accurate method of debugging is

doing by hand with pencil and paper what the computer does electronically. You'll learn how to analyze a code sequence and follow the changing states of indexes and variables. Sometimes it is necessary to examine information stored inside the computer. Chapter 3 explains how to use print statements in an effort to track down even the most elusive bug.

And once in awhile, in spite of your best effort, a bug will successfully avoid correction. Most often you'll run into this situation in simulation and game programs where the program is more comples rather than a straightforwarrd series of events. In Chapter 4 you'll learn how to delvelop and integrate a patch to get around a problem that simply evades solution.

In Chapter 5 you'll learn where and how each debugging tool can be used most effectively. You'll go through a typical debugging effort and observe how a programmer uses the tools available to achieve an accurate cure for each situation in the shortest possible time.

At some point, you'll probably face a situation where a new or seldom-used program will not run on your computer. Where no bus appear obvious^ you may save frustrating hours by making sure your hardware is working properly. Chapter 6 explains the operation of the most popular peripheral units in an effort to lead you to a bug that is not in the programming. If you're a hobbyist or a beginning programmer, this book should serve you well as a basic handbook on debugging. You'll find more advanced techniques on debugging in *Software Debugging for Microcomputers* and *Personal Computing* (Reston Publishing Co.). The material in this book was drawn from these volumes.

How to Debug Your Personal Computer

1

Debugging the Flowchart

Debugging is not always performed after the fact; time spent catching potential bugs in the preliminary stage of program development could mean hours of time saved later on during the actual testing and shakedown stage.

The first step in writing a computer program is to list on paper all those tasks the program is expected to accomplish. This helps to organize thoughts and clarify objectives, and it gives the programmer a direction and a goal.

FLOWCHARTING THE EXISTING PROGRAM

Drawing up a flowchart first—before composing any lines of code—is a good way to catch potential errors in program flow, which should be welcome news since such bugs are usually the hardest to locate.

By the same token, generating a flowchart for an already written program is an excellent way to track down elusive bugs. The secret here is that since flowcharts are graphic representations of the flow of the program, bugs which can get lost in a jumble of statement numbers and

conditional branches become glaringly obvious once drawn in picture form.

Take as an example the following program. Even though it is well documented, it is long enough and complicated enough that unless we resort to graphing it out, we might never discover the bug.

```
10 REM TRIP CONTROL PROGRAM FOR
20 REM GREAT CENTRAL MODEL RAILROAD
30 PRINT "WELCOME ABOARD THE GREAT CENTRAL"
40 PRINT "MODEL RAILWAY. HOW MANY"
50 PRINT "ROUND TRIPS TODAY?"
60 INPUT T
70 REM T IS THE TRIP COUNTER
80 REM LOOP TO DETERMINE TRAIN LOCATION
90 REM DATA PORTS ARE
100 REM 1=IN STATION
110 REM 2=NEARING SW 1 FROM INNER LOOP
120 REM 3=NEARING TIGHT TURN
130 REM 4=NEARING STRAIGHTAWAY
140 REM 5=NEARING SW 2
150 REM 6=NEARING X-ING
160 REM 7=ON X-ING
170 REM 8=NEARING SW 1 FROM OUTER LOOP
180 REM 9=NEARING STATION
190 REM 10=THROTTLE
200 REM 11=X-ING SIGNAL
210 REM 12=SW1
220 REM 13=SW2
225 LET N=1
230 FOR I=1 TO T
240 LET N=-N
250 LET A=INP(1)
260 IF A=255 THEN GOSUB 440
270 LET B=INP(2)
280 IF B=255 THEN GOSUB 530
290 LET C=INP(3)
300 IF C=255 THEN GOSUB 660
310 LET D=INP(4)
320 IF D=255 THEN GOSUB 750
```

```
330 LET E=INP(5)
340 IF E=255 THEN GOSUB 840
350 LET F=INP(6)
360 IF F=255 THEN GOSUB 950
370 LET G=INP(8)
380 IF G=255 THEN GOSUB 1080
390 LET H=INPUT(9)
400 IF H=255 THEN GOTO 1120
410 REM POSITION SENSOR ACTIVATED PUTS
420 REM ALL ONES (255) ON DATA LINES
430 GOTO 250
434 NEXT I
436 GOTO 1280
440 PRINT "ALL ABOARD"
450 REM WAIT FOR PASSENGERS TO BOARD
460 PAUSE 40
470 REM ACCELERATE OUT OF STATION TO SPEED 5
480 FOR J=1 TO 5
490 OUT 10, J
500 PAUSE 10
510 NEXT J
520 RETURN
530 REM NEARING SW 1
540 REM PULSE SW 1, ALL ONES
550 REM PULSES SW 1 TO INSIDE CIRCLE
560 OUT 12, 255
570 REM SLOW TRAIN ONE STEP
580 OUT 10, 4
590 REM HAVE WE CLEARED SENSOR 2 YET?
600 LET R=INP(2)
610 IF R>0 THEN GOTO 600
620 REM WAIT TO CLEAR SW 1 AND ACCEL.
630 PAUSE 10
640 OUT 10, 5
650 RETURN
660 REM NEARING RIGHT TURN
670 REM CUT TRAIN SPEED TO 2
680 FOR J=1 TO 3
690 LET S=INP(10)
```

```
700 LET S=S-1
710 OUT 10, S
720 PAUSE 10
730 NEXT J
740 RETURN
750 REM NEARING STRAIGHTAWAY
760 REM INCREASE SPEED TO 10
770 FOR J=1 TO 8
780 LET S=INP(10)
790 LET S=S+1
800 OUT 10, S
810 PAUSE 10
820 NEXT J
830 RETURN
840 REM NEARING SW 2
850 REM SET SW FOR OUTSIDE LOOP
860 IF T<0 THEN OUT 13, 0
870 REM CUT SPEED TO 5
880 FOR J=1 TO 5
890 LET S=INP(10)
900 LET S=S-1
910 OUT 10, S
920 PAUSE 10
930 NEXT J
940 RETURN
950 REM NEARING X-ING
960 REM SET WARNING FLASHER SWITCH
970 REM PORT 11 IS X-ING FLASHER
980 LET K=1
990 IF K>0 THEN GOTO 1020
1000 REM FLASHER ON
1010 OUT 11, 255
1020 PAUSE 5
1030 LET K=-K
1040 REM TEST IF WE HAVE CLEARED X-ING
1050 LET S=INP(7)
1060 IF S>0 THEN GOTO 990
1070 RETURN
1080 REM NEARING SW 1 FROM OUTSIDE
```

```
1090 REM SET SW 1 TO OUTSIDE
1100 OUT 12, 0
1110 RETURN
1120 REM NEARING STATION
1130 PRINT "NOW APPROACHING SMALLTOWN STATION"
1140 PRINT "SMALLTOWN, USA"
1150 REM SLOW TRAIN TO 1
1160 FOR J=1 TO 4
1170 LET S=INP(10)
1180 LET S=S-1
1190 OUT 10, S
1200 PAUSE 10
1210 NEXT J
1220 REM CHECK WHEN TRAIN MAKES STATION
1230 LET K=INP(1)
1240 IF K=0 THEN GOTO 1230
1250 REM STOP TRAIN
1260 OUT 10, 0
1270 GOTO 434
1280 PRINT "END OF TODAY'S RUN"
1290 END
```

At first glance, 129 lines of code may seem overpowering; but we will see that it only helps to point up the usefulness of flowcharting as a debugging tool.

Before we begin to generate our flowchart, we can learn a number of things about the program just by inspection. The program has been well documented, and we will use this documentation to our advantage as we go along.

We can tell, for instance, that the program seems to be divided into three distinct parts. The first part consists mainly of remarks explaining what the program is and what its assorted variables stand for. The second part is the main program (only about 15 lines long); and the third part consists of all those subroutines that were referenced in the main program.

Something which the program does not list, but which would be helpful to us as we try to visualize what is taking place, is a diagram of the track layout. The Great Central model railroad of the referenced program is shown schematically in Fig. 1-1.

The track has an inner route and an outer route. Both routes share a common side, the one which includes the tight turn and the straightaway. The inner loop passes by the station, where it must stop long enough for

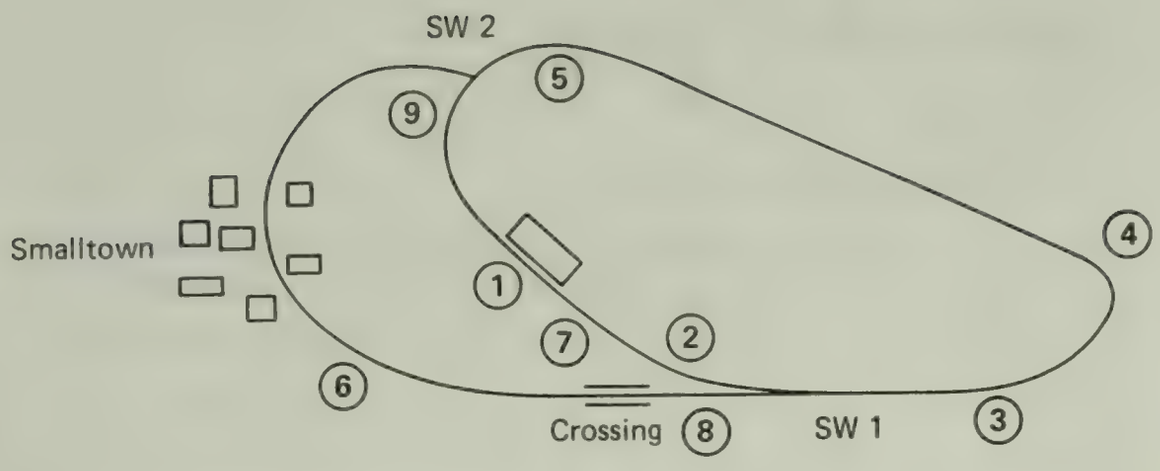


FIG. 1-1. Layout of the Great Central model railroad.

passengers to board and disembark. The outer loop carries the train through the manufacturing and business section of Smalltown.

The program was written to monitor and control the speed and location of the Great Central model railroad as it alternately traverses first the inner and then the outer loop of its right-of-way.

In order to monitor the train's location, sensors have been set up at various points along the track. Actuators, which respond to a signal of 0 (all zeros) or 255 (all ones) on their data lines, have been installed on both switches and on the railroad crossing warning flashers.

We may assume that all required analog-to-digital interfacing has been properly attended to, and as a result the only bugs (there are two) are attributable to software.

We begin with the first part of the program, the explanatory section. Flowcharting this (Fig. 1-2) is trivial, since the computer does not actually perform very much.

The second part of the program, the main part, is also not difficult to graph. Each sensor in turn is interrogated; and if the sensor transmits an all-ones activated signal (numerical value equal to 255), then the program branches to the appropriate subroutine before it continues the cycling sensor interrogation.

We can draw this portion of the flowchart as shown in Fig. 1-3.

DOCUMENTING

Since we were only trying to document the block of coding beginning at line 225 and ending at line 436, there were a few places that we were forced to leave blank. Specifically, we know that there is a NEXT statement to bracket the FOR statement:

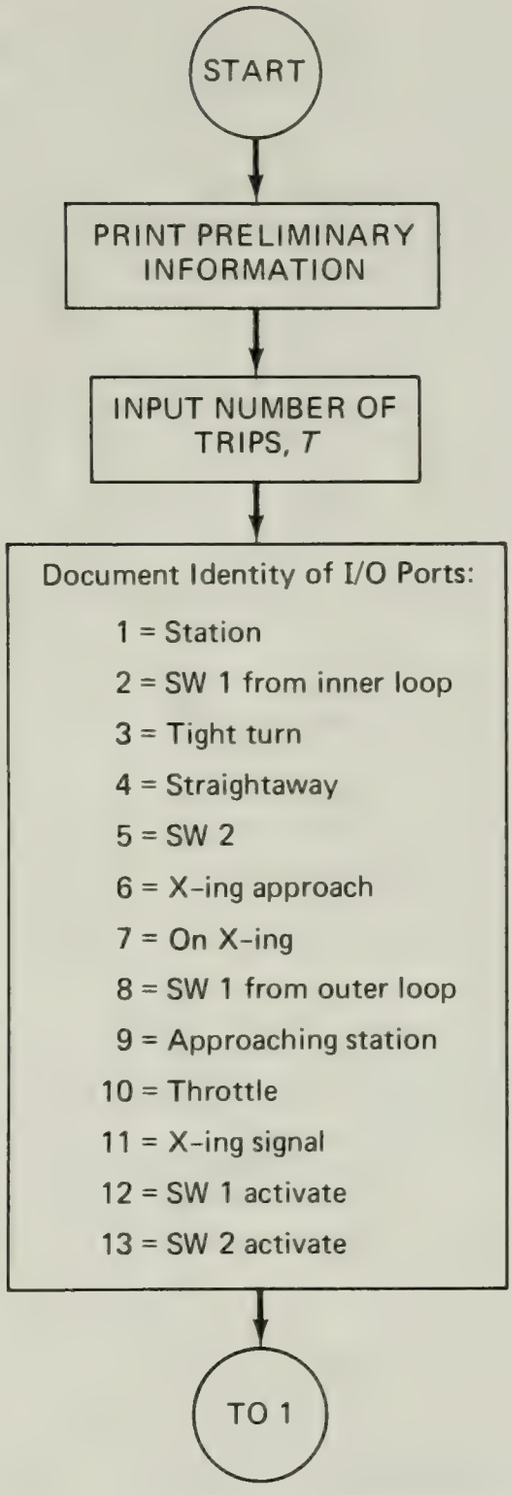


FIG. 1-2. Flowcharting the explanatory segment of the railroad program.

```
230 FOR I=1 TO T
    ≀
    ≀
434 NEXT I
```

But there is no statement in the block of coding we have just examined which sends the program down to line 434 so that the loop may be

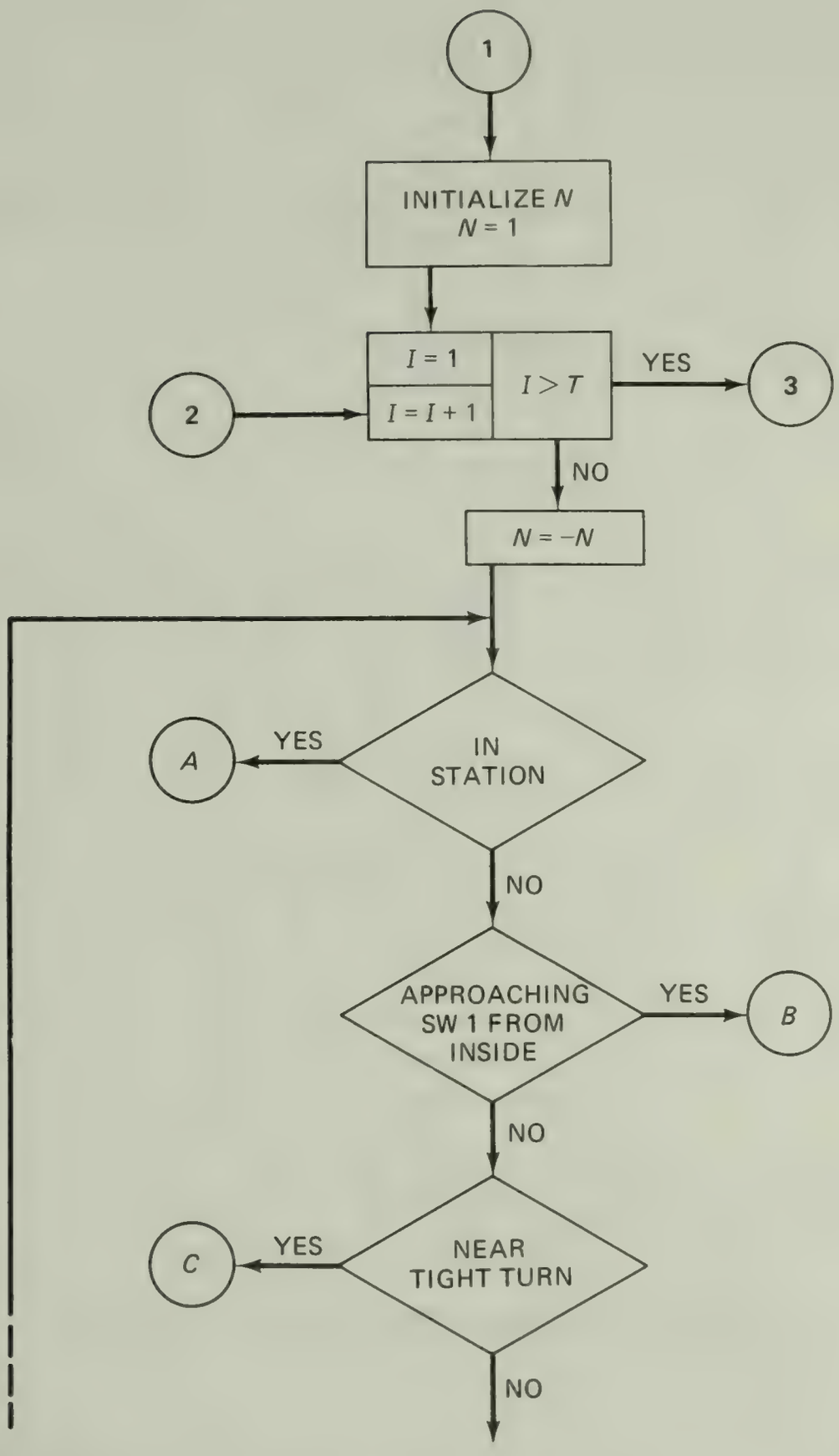


FIG. 1-3. Flowchart of the model railroad's main program.

incremented. We assume, then, that unless this is the bug, statement 434 must be entered from some other point in the program. We signify this by having a transfer symbol feed into the incrementing section of the loop symbol.

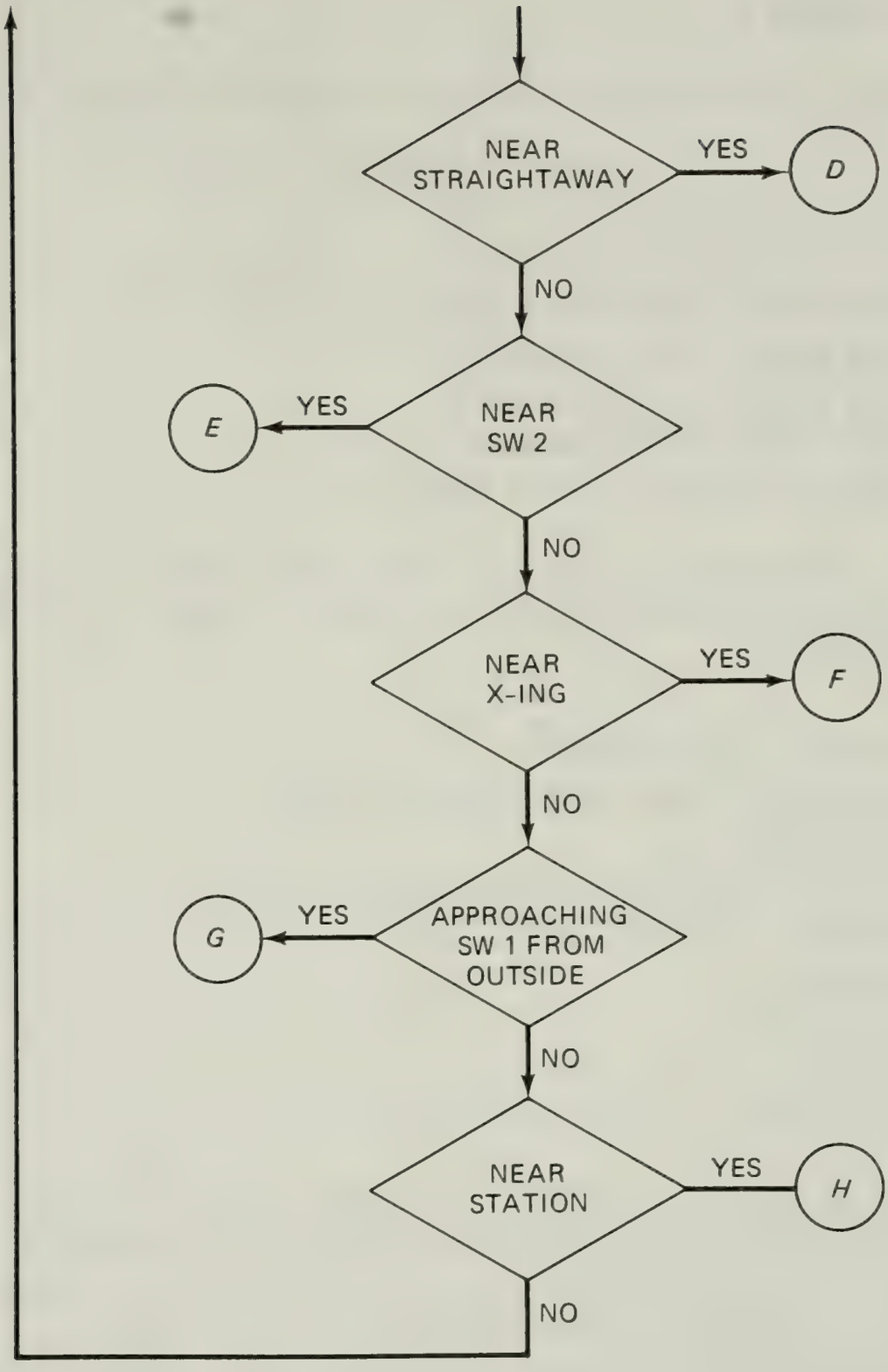


FIG. 1-3. (Cont'd.)

We note that the eight conditional branches themselves form a loop. For each IF statement, if the condition is met, program flow is transferred to a subroutine. If the first condition is not met, the next one is tried, and so on, apparently forever. This could be a bug: the program looks as though it is caught in an endless loop. Rather than jump to any conclusions, however, we should finish our flowcharting of the complete program.

SUB-ROUTINES

There are seven subroutines, beginning at lines 440, 530, 660, 750, 840, 950, and 1080:

```
260 IF A=255 THEN GOSUB 440
280 IF B=255 THEN GOSUB 530
300 IF C=255 THEN GOSUB 660
320 IF D=255 THEN GOSUB 750
340 IF E=255 THEN GOSUB 840
360 IF F=255 THEN GOSUB 950
380 IF G=255 THEN GOSUB 1080
```

We will attempt to flowchart each in turn. Subroutine A, which begins at line 440, is entered if status switch 1, located in the station, reads positive:

```
440 PRINT "ALL ABOARD"
450 REM WAIT FOR PASSENGERS TO BOARD
460 PAUSE 40
470 REM ACCELERATE OUT OF STATION TO SPEED 5
480 FOR J=1 TO 5
490 OUT 10, J
500 PAUSE 10
510 NEXT J
520 RETURN
```

This subroutine prints an "all aboard" message and waits long enough for passengers to board the train. Then over a period of five seconds, the train pulls away from the station and increases its speed to 5 points (of a possible 10). We would diagram it as in Fig. 1-4.

Subroutine B, which begins at line 530 and ends at line 650, is called when the train is near switch 1, approaching from the inside curve. If the train is not to derail, the switch must be thrown up so that the tracks of the switch line up with the tracks of the inside loop. For both switch 1 and switch 2, the all-ones command (255) sets the switch to line up with the inside loop.

The coding for subroutine B looks like this:

```
530 REM NEARING SW 1
540 REM PULSE SW 1, ALL ONES
```

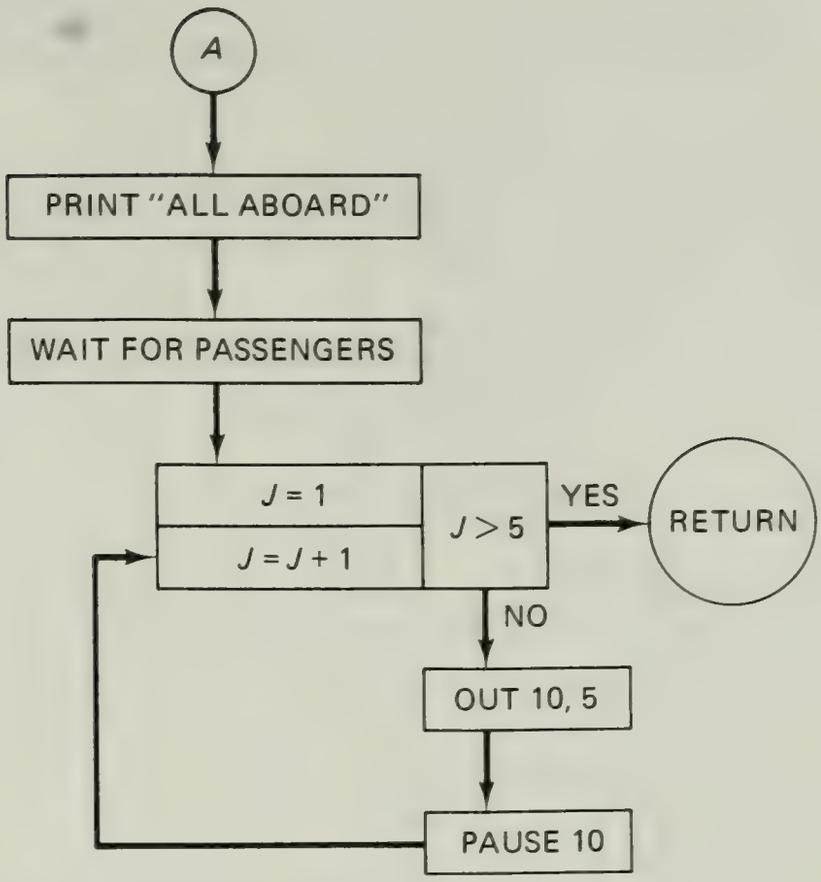


FIG. 1-4. Flowchart for "all aboard" subroutine.

```
550 REM PULSES SW 1 TO INSIDE CIRCLE
560 OUT 12, 255
570 REM SLOW TRAIN ONE STEP
580 OUT 10, 4
590 REM HAVE WE CLEARED SENSOR 2 YET?
600 LET R=INP(2)
610 IF R>0 THEN GOTO 600
620 REM WAIT TO CLEAR SW 1 AND ACCEL.
630 PAUSE 10
640 OUT 10, 5
650 RETURN
```

The flowchart for those 13 program lines that make up subroutine B has the appearance of Fig. 1-5.

The first thing that this subroutine does is to set the switch to the proper orientation so that when the train arrives at the junction it will move smoothly out onto the main loop and not become derailed.

We note that the "mini-loop" between steps three and four, where the subroutine cycles through, checks if station 2 reports all clear. Once the sensor reports clear, the train is given another second of traveling

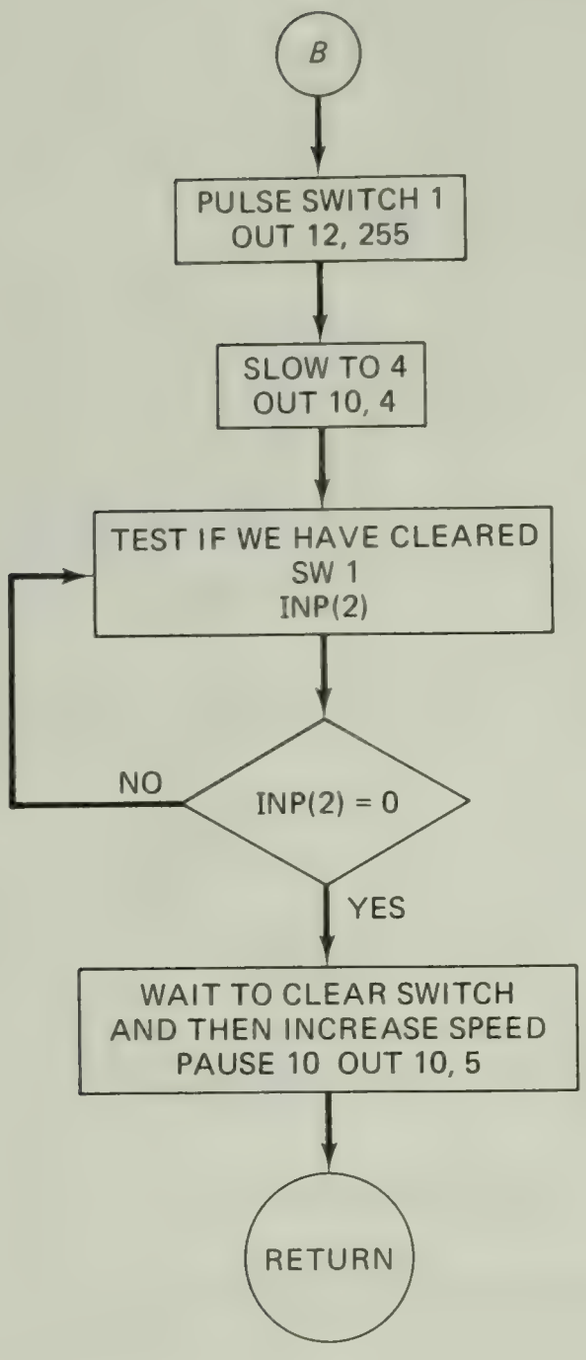


FIG. 1-5. Flowchart for subroutine B.

time (the argument for the PAUSE command is in tenths of a second) to allow it to clear the switch. Then the throttle is increased by a step.

Subroutine C slows the train enough to allow it to safely negotiate the tight turn just as subroutine D opens up the throttle to full power as the train enters the straightaway. The two subroutines are therefore nearly identical in construction and content, as this side-by-side comparison shows:

Subroutine C	*Subroutine D*
660 REM NEARING TIGHT TURN	750 REM NEARING STRAIGHTAWAY

Subroutine C

```
670 REM CUT TRAIN SPEED
  TO 2
680 FOR J=1 TO 3
690 LET S=INP(10)
700 LET S=S-1
710 OUT 10, S
720 PAUSE 10
730 NEXT J
740 RETURN
```

Subroutine D

```
760 REM INCREASE SPEED
  TO 10
770 FOR J=1 TO 8
780 LET S=INP(10)
790 LET S=S+1
800 OUT 10, S
810 PAUSE 10
820 NEXT J
830 RETURN
```

Each subroutine is essentially a short loop which uses the index variable to change the relative setting of the throttle (I/O PORT 10). In both cases, a supposition is made concerning the speed of the train as it enters the field of each sensor and thus as the program enters the domain of each subroutine. Subroutine C assumes the train is traveling at throttle setting 5, so that it can reduce the setting by 3 points to achieve a setting of 2. Subroutine D assumes that it will in all cases be called after C and none else, so that the throttle setting will be at 2 and will need an increase of 8 points to bring it up to full open.

For the moment, we merely take note of this information, but in the future it may help track down a bug, should the assumptions about incoming train speed prove false.

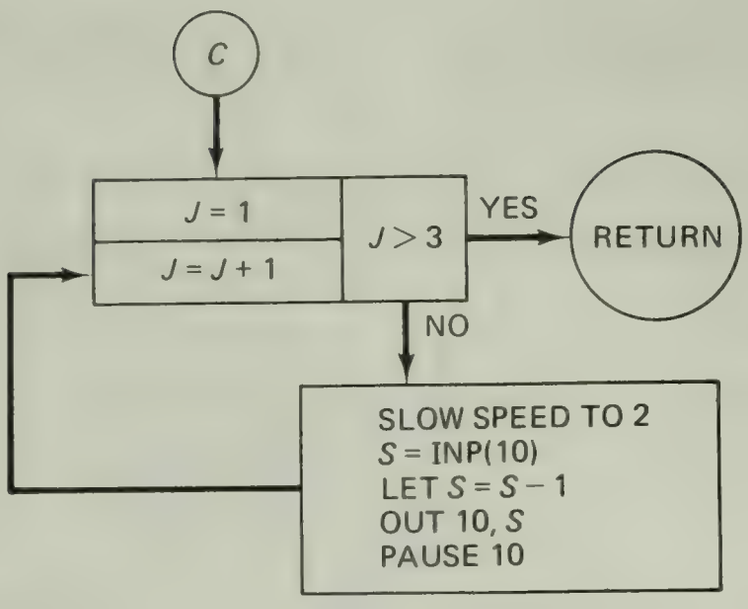
Flowcharting subroutines C and D is not difficult. The form is depicted in Fig. 1-6.

Subroutine E has two important tasks to accomplish. It must set switch 2 for the proper loop, and it must decrease the train's speed to a throttle setting of 5. No matter which fork in the track the train takes, through town or to the station, the throttle must be at 5. This is done easily enough with a block of coding nearly identical to subroutine C.

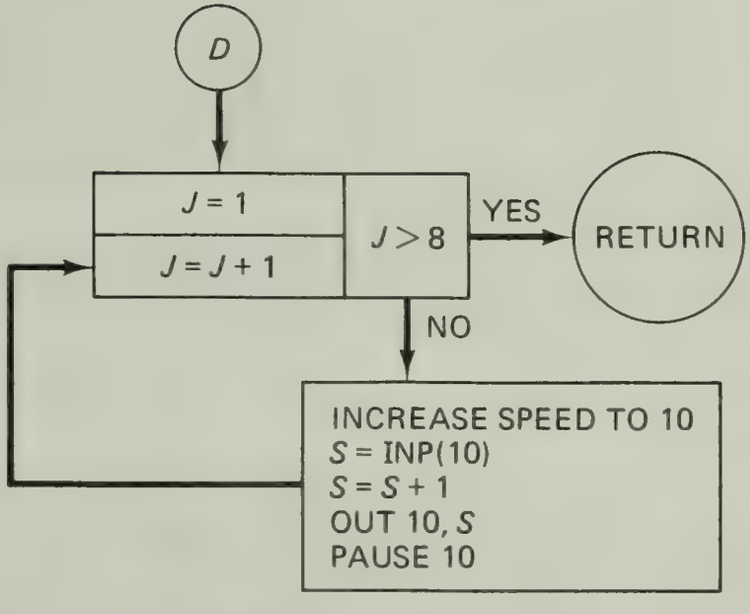
Setting the switch for the proper loop is a little harder. The program must have some way of knowing if it is on the first or the second lap of its two-lap cycle. On the first lap, the train will be coming from the station, and so we would want it to be switched to the townbound loop. On the second lap, the train must be shunted off to the station to allow the passengers to leave and to board.

The coding is as follows, and the flowchart is shown in Fig. 1-7.

```
840 REM NEARING SW 2
850 REM SET SW FOR OUTSIDE LOOP
860 IF T<0 THEN OUT 13, 0
870 REM CUT SPEED TO 5
```



(A)



(B)

FIG. 1-6. Flowchart for (A) subroutine C and (B) subroutine D.

```
880 FOR J=1 TO 5
890 LET S=INP(10)
900 LET S=S-1
910 OUT 10, S
920 PAUSE 10
930 NEXT J
940 RETURN
```

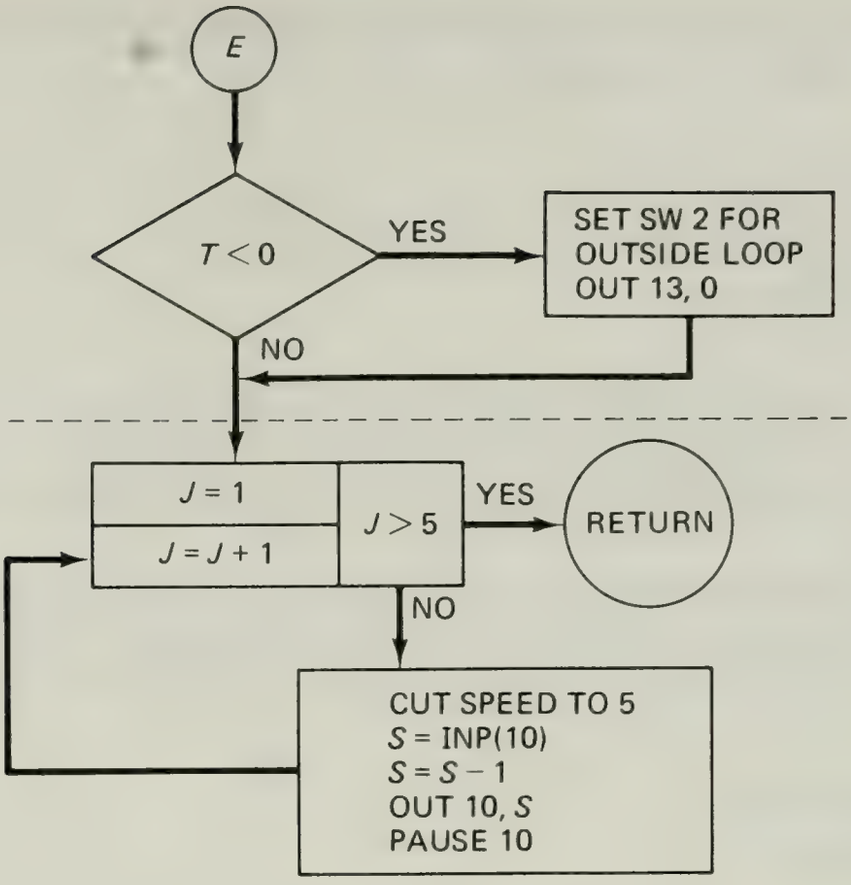


FIG. 1-7. Flowchart for setting the switch for proper loops.

We can divide subroutine E into two parts. The part below the dotted line in Fig. 1-7 is very similar to subroutine C, just as we said it would be. The section of subroutine E above the dotted line, however, has problems. We see that the program tests on *T*, and if *T* is less than zero it pulses switch 2 and causes the track to line up with the outside loop. But *T* is the variable used to index the number of round trips. Recall:

```
60 INPUT T
70 REM T IS THE TRIP COUNTER
```

Also, *T* is never made negative anywhere in the program. This being true, the test of subroutine E would never be met, and SW 2 would never be set properly. Depending upon how switch 2 was set when power was applied, the train would either run in perpetual circles through town or through the station.

We can guess that the program is trying to test which lap it is on, and that it is trying to use that information to decide whether or not to pulse the switch.

If *T* is not the correct variable, what is? The variable *N* is initialized

to a positive 1; and then once inside the trip loop, it flips back and forth between positive and negative:

```
225 LET N=1
230 FOR I=1 TO T
240 LET N=-N
```

If we were to key on *N*, what would be the outcome? The first time through, *N* would be less than zero, and the test would be met. As a result, SW 2 would be set to the outside loop, and the train would go into Smalltown.

The subroutine would then return control to the main program. But, as we learned when we flowcharted the main program, there is no provision for getting the code down to line 434, where the loop is incremented:

```
430 GOTO 250
434 NEXT I
```

As a result, *N* would remain forever negative and the train would continue to follow the outer loop, never returning to the station.

Line 250 begins the loop which queries each sensor in turn to determine the train's location. We might try moving the sign-change coding for *N* inside of this loop:

```
250 LET A=INP(1)
255 LET N=-N
```

Then every time the main program jumps back to line 250 to begin another status search, the sign of *N* is changed. Unfortunately, this loop is executed many times each second, since the computer can process instructions far faster than the train can travel from sensor to sensor (which explains why so many PAUSE statements have been included). Consequently, the status of *N* would be merely a matter of chance and would not be a reliable index for switch 2.

We can make an assumption, however, which should be true in all cases. Each time the train leaves the station it pulses switch 1 to the inside circle. It does this without asking the status of switch 1 beforehand. Switch 1 is also unconditionally set to the outer loop whenever the train reaches the crossing. It would appear, then, that by checking the status of switch 1, we may determine what the status of switch 2 ought to be. This is reflected in the revised flowchart portion depicted in Fig. 1-8.

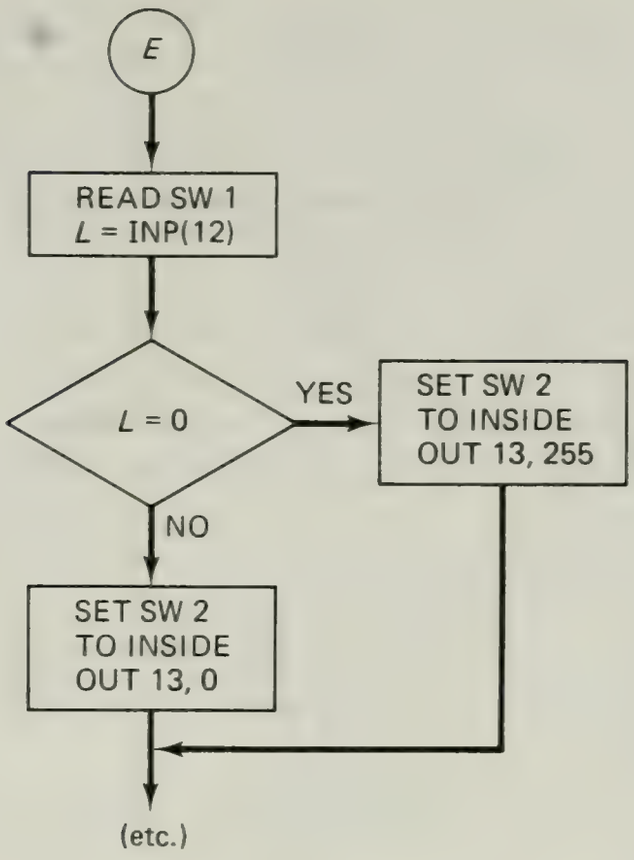


FIG. 1-8. Flowchart to determine the status of switch 1, which tells us what switch 2 should be.

Recoding of that section of the subroutine might look something like:

```
840 REM NEARING SW 2
845 REM TEST STATUS OF SW 1
850 LET L=INP(12)
852 REM DEFAULT SW 2 TO OUTSIDE
854 OUT 13, 0
860 IF L=0 THEN OUT 13, 255
```

Subroutine F is concerned with activating the warning signal at the train crossing. For this task the program must flash a warning light as long as the train is in the crossing. The flowchart for subroutine F is shown in Fig. 1-9; it is coded as follows:

```
950 REM NEARING X-ING
960 REM SET WARNING FLASHER SWITCH
970 REM PORT 11 IS X-ING FLASHER
980 LET K=1
990 IF K>0 THEN GOTO 1020
1000 REM FLASHER ON
```

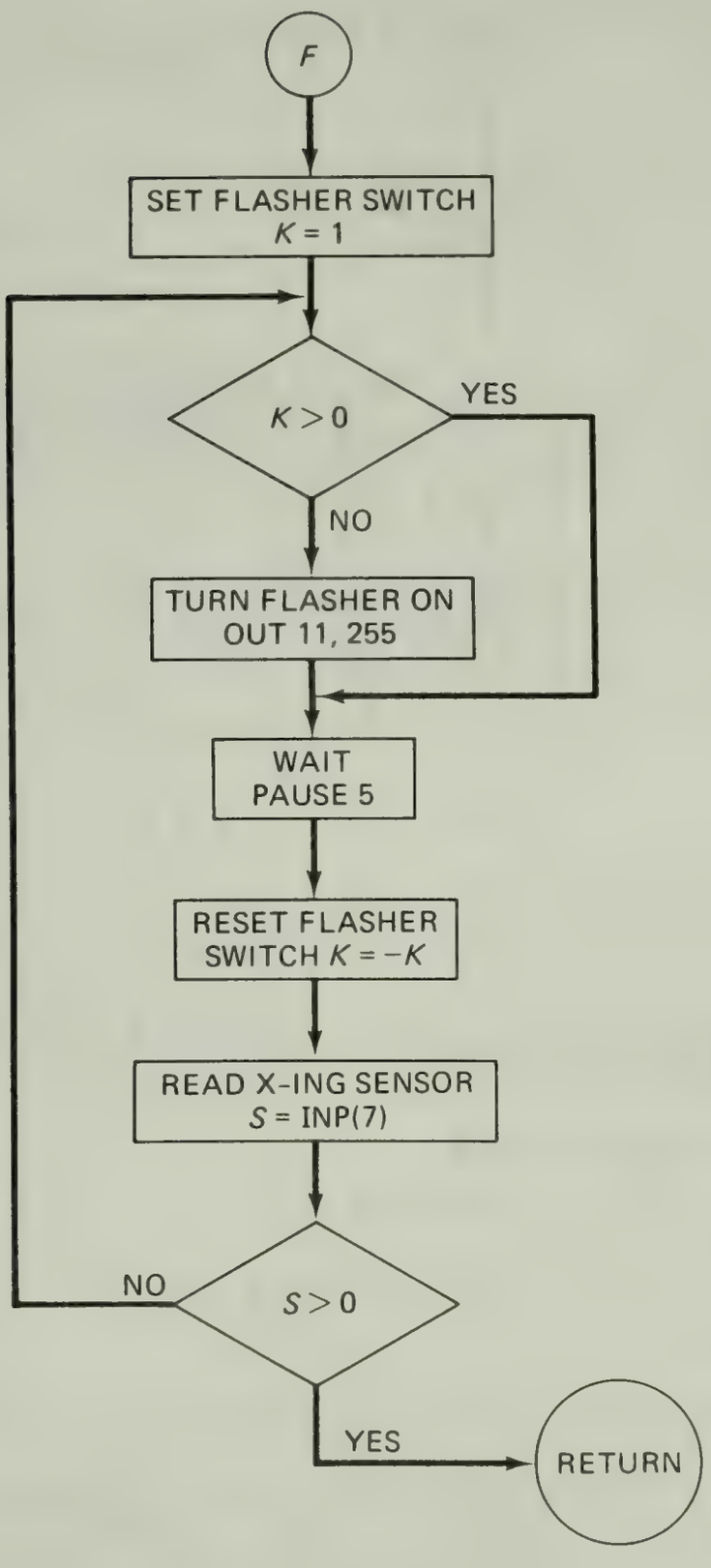


FIG. 1-9. Flowchart for subroutine F.

```
1010 OUT 11, 255
1020 PAUSE 5
1030 LET K=-K
1040 REM TEST IF WE HAVE CLEARED X-ING
1050 LET S=INP(7)
1060 IF S>0 THEN GOTO 990
1070 RETURN
```

Using words to describe what the flowchart tells us in pictures, let's see what we have. The program enters subroutine F, and a switch variable *K* is initialized to 1. *K* is then tested to see if it is greater than zero. It is, so the program pauses, changes the sign of *K* to negative, and interrogates sensor 7 to see if the train has left the crossing yet. The train has not, so the program jumps back to the test of *K*. This time, *K* is negative, and consequently the warning light is turned on. Again the pause, change of sign, interrogation of sensor 7, and jump back to the test of *K*. Now the sign of *K* is back to positive, so the program . . . a bug! The second one we discovered using this technique. The warning light never gets turned off. It is supposed to flash on and off, but as we can see using the flowchart, nothing of the sort takes place.

A logical fix for this bug would be to modify the flowchart as shown in Fig. 1-10.

Subroutine G is trivial in that it need only pulse switch 1 to line up with the outer circle whenever sensor 8 is activated:

```
1080 REM NEARING SW 1 FROM OUTSIDE
1090 REM SET SW 1 TO OUTSIDE
1100 OUT 12, 0
1110 RETURN
```

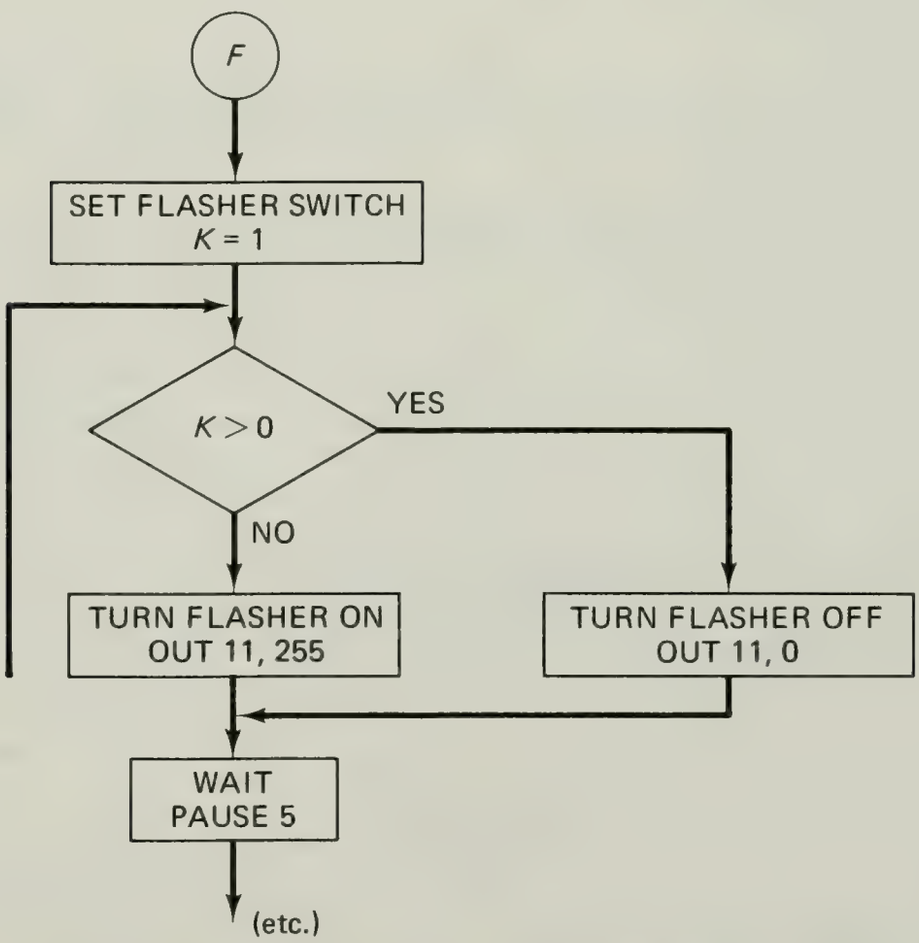


FIG. 1-10. Flowchart modification to generate flashing model light.

Its flowchart, equally simple, is shown in Fig. 1-11.

The coding which begins at line 1120 and ends at line 1270—what we call subroutine H—is not technically a subroutine in that it does not end with a RETURN statement; rather, it terminates with a GOTO:

```
1270 GOTO 434
```

At least we have found how the program closes the main loop. As the train approaches the station, it is slowed down; the structure of this part of subroutine H is similar to all of the other speed-change subroutines, and so it should be familiar to us by now. Once the train reaches the station (as reported by sensor 1), it is brought to a dead stop and the main program loop is incremented. The coding is as follows:

```
1120 REM NEARING STATION
1130 PRINT "NOW APPROACHING SMALLTOWN STATION"
1140 PRINT "SMALLTOWN, USA"
1150 REM SLOW TRAIN TO 1
1160 FOR J=1 TO 4
1170 LET S=INP(10)
1180 LET S=S-1
1190 OUT 10, S
1200 PAUSE 10
1210 NEXT J
1220 REM CHECK WHEN TRAIN MAKES STATION
1230 LET K=INP(I)
1240 IF K=0 THEN GOTO 1230
1250 REM STOP TRAIN
```

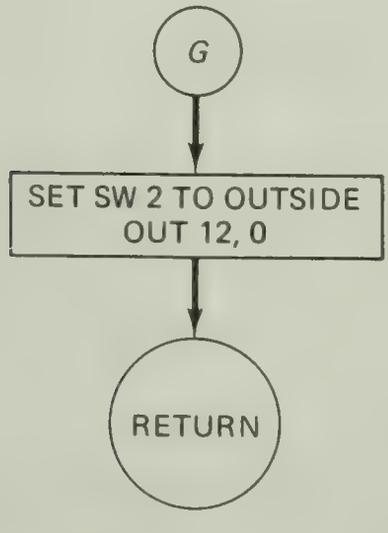


FIG. 1-11. Flowchart for subroutine G.

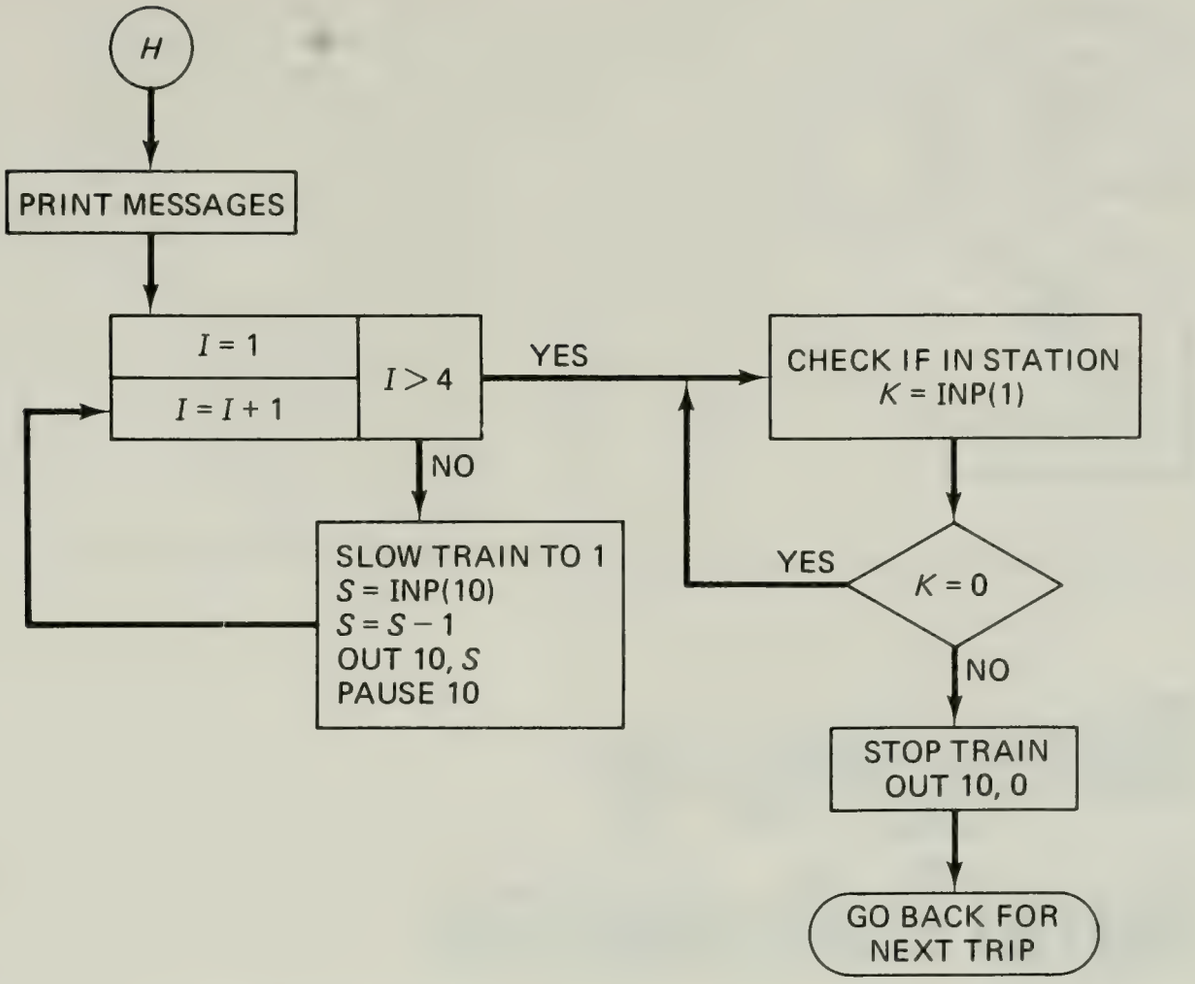


FIG. 1-12. Flowchart for subroutine H.

```
1260 OUT 10, 0
1270 GOTO 434
```

The flowchart derived from that coding is shown in Fig. 1-12.

There do not appear to be any bugs in this last block of coding. On paper, at least, the program is debugged. Once the program is put into use, other bugs might very well appear. For instance, perhaps the train accelerates or decelerates too quickly. Or perhaps because of the relative placement of the position sensors, the switches do not react quickly enough. Both bugs would be related to the same problem: an incorrectly specified PAUSE statement. Unfortunately, the program is full of PAUSE statements. The program *is* well documented, however, and this is a real help when trying to debut. But the availability of flowcharts would help make the solution for such bugs apparent at a glance.

2

Debugging by Hand Calculation

One of the best ways to debug a program is to play computer. This method often takes more time than others, and is not as easy as tracing program execution with print statements, but the attention to detail that playing computer requires of the programmer forces a clear understanding of what a particular piece of code is doing.

As an example, we will try to debug the following program, the flowchart for which is shown in Fig. 2-1.

```
10 REM PROGRAM IPOWR
20 REM FINDS THE VALUE OF A TO THE B
30 REM POWER WHERE A AND B ARE BOTH
40 REM INTEGERS AND THE RESULT IS
50 REM ACCURATE TO 100 PLACES
60 REM SET UP BUFFERS
70 DIM A1(100), B1(100), A2(100)
80 DIM C1(10), B2(100)
90 LET J=0
```

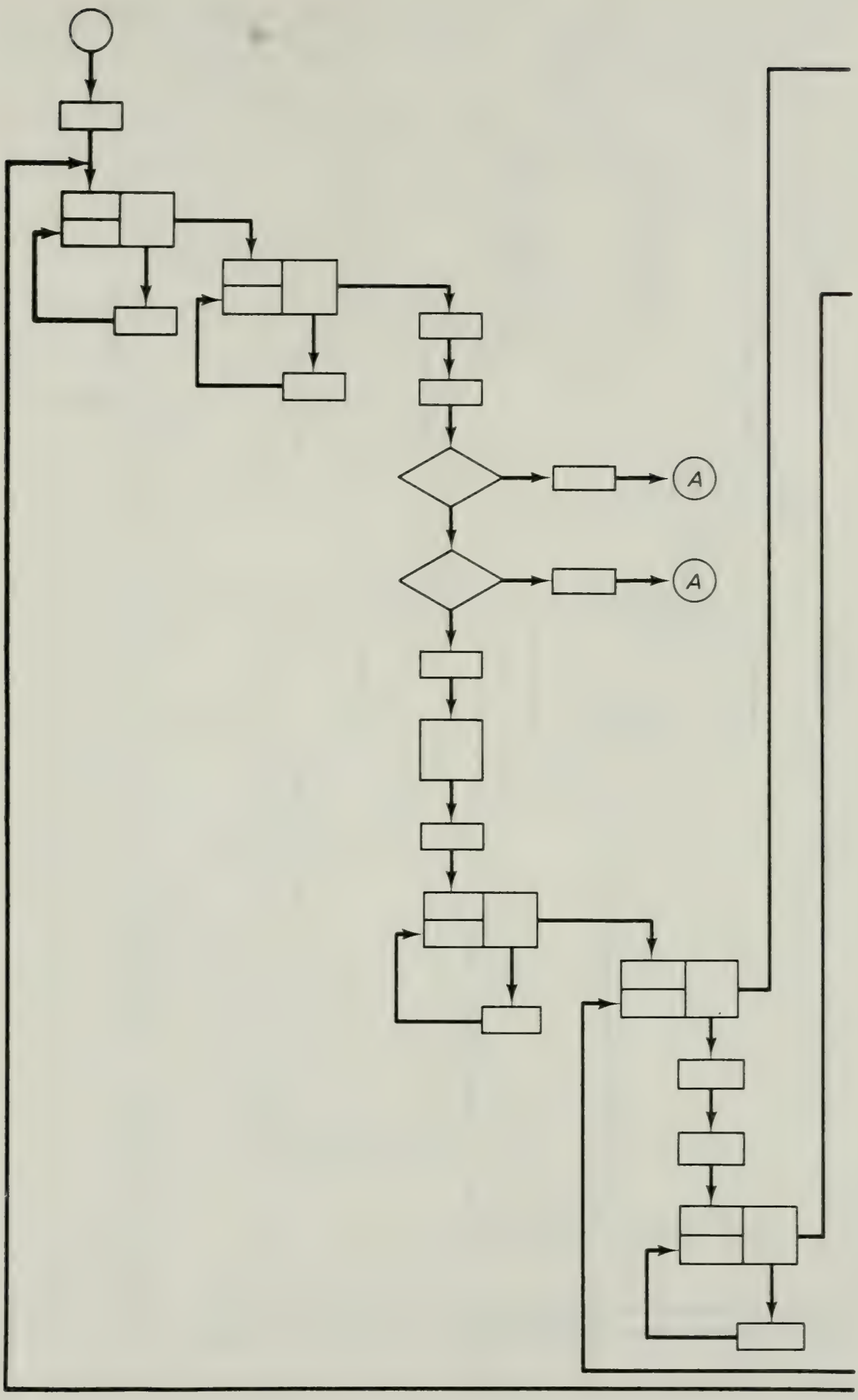


FIG. 2-1. Flowchart from program IPOWR, general overview. The individual blocks are identified in subsequent illustrations.

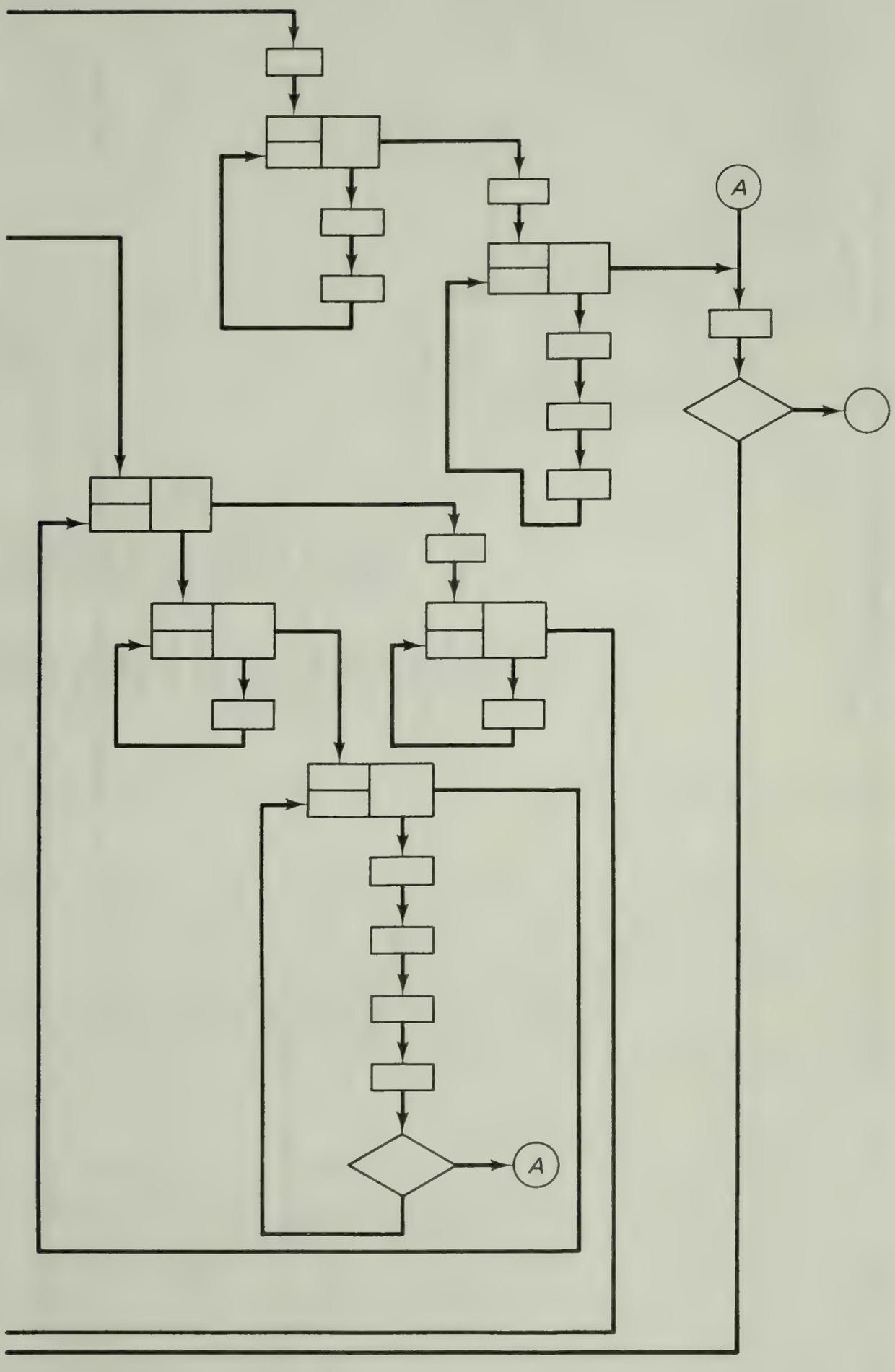


FIG. 2-1. (Cont'd.)

```
100 REM J=ROOT DIGIT COUNTER
110 REM CLEAR BUFFERS
120 FOR I=1 TO 100
130 LET A1(I)=0
140 LET B1(I)=0
150 LET A2(I)=0
160 LET B2(I)=0
170 NEXT I
180 REM A1=ROOT STORAGE BY DIGIT
190 REM B1=CARRY ARRAY
200 REM A2=PRINT ARRAY
210 REM B2=TEMP RESULT ARRAY
220 FOR I=1 TO 10
230 LET C1(I)=0
240 NEXT I
250 REM C1=MULTIPLIER DIGIT STORAGE
260 INPUT R, E
270 PRINT "& K"
280 PRINT "WORKING . . ."
290 REM R=ROOT
300 REM E=EXPONENT
310 IF R=0 THEN GOTO 1290
320 IF E=0 THEN GOTO 1310
330 LET R1=R
340 LET R2=R
350 REM R1=INCREMENTAL FLOATING POINT RESULT
360 REM R2=CONSTANT MULTIPLIER
370 REM
380 REM REDUCE E BY 1 SINCE NO. OF
390 REM MULTIPLICATIONS IS ONE LESS THAN
400 REM VALUE OF EXPONENT
410 LET E=E-1
420 LET J2=E
430 REM J2=EXPONENT STORAGE
440 REM INCREMENT ROOT DIGIT COUNTER
450 LET J=J+1
460 REM TRANSFER ROOT TO A1 ARRAY
470 REM A DIGIT AT A TIME
480 LET A1(J)=R-(10*INT(R/10))
```

```
490 LET R=INT(R/10)
500 REM TEST IF ALL DIGITS ARE ENTERED
510 IF R>0 THEN GOTO 450
520 REM SET ROOT AND MULT. ARRAYS EQUAL
530 FOR I2=1 TO J
540 LET C1(I2)=A1(I2)
550 NEXT I2
560 REM BEGIN LOOP ON EXPONENT
570 FOR I=1 TO E
580 REM MULT. INCREASING RESULT BY ROOT
590 LET R1=R1*R2
600 REM HOW MANY DIGITS IN RESULT?
610 LET P=INT(LOG10(R1))+1
620 REM P=NUMBER OF PLACES
630 REM CLEAR TEMP RESULT ARRAY
640 FOR J1=1 TO 100
650 LET B2(J1)=0
660 NEXT J1
670 REM BEGIN LOOP ON MULTIPLIER DIGIT COUNT
680 FOR M1=1 TO J
690 REM CLEAR CARRY ARRAY
700 FOR N=1 TO 100
710 LET B1(N)=0
720 NEXT N
730 REM BEGIN LOOP ON MULTIPLICAND DIGIT COUNT
740 FOR M2=1 TO P
770 REM DO ONE DIGIT MULTIPLY, MULTIPLIER*
780 REM MULTIPLICAND+CARRY
790 LET T1=A1(M2)*C1(M1)+B1(M2)
800 REM T1=TEMPORARY RESULT
810 REM WHAT IS OUR DIGIT POSITION?
820 LET D=M1-1+M2
830 REM ADD RESULT OF THIS MULTIPLICATION
840 REM TO PREVIOUS RESULT
850 REM (T1-10*(INT(T1/10)))=UNITS DIGIT OF RESULT
860 REM B2=PREVIOUS RESULT DIGIT
870 LET T2=(T1-10*(INT(T1/10)))+B2(D)
880 REM BREAK THIS RESULT INTO UNITS AND TENS
890 REM FOR THE UNITS;
```

```
900 LET B2(D)=T2-10*(INT(T2/10))
910 REM FOR THE TENS;
920 LET B2(D+1)-INT(T2/10)
930 REM HAVE WE RUN OUT OF ROOM?
940 IF D+1=100 THEN GOTO 1270
950 REM FIND NEW CARRY DIGIT
960 LET B1(M2+1)=INT(T1/10)
970 NEXT M2
980 NEXT M1
990 REM UPDATE MULTIPLICAND ARRAY
1000 FOR N=1 TO 100
1010 LET A1(N)=B2(N)
1020 NEXT N
1060 NEXT I
1070 REM REVERSE ORDER OF RESULT ARRAY TO PRINT
1080 FOR N=1 TO P
1090 LET C=(P+1)-N
1100 LET A2(N)=A1(C)
1110 NEXT N
1120 REM PREPARE TO DISPLAY ANSWER
1130 REM RESTORE EXPONENT
1140 LET E=E+1
1150 CURSOR 6, 0
1160 PRINT "FOR THE EXPRESSION"; R2; "TO THE";
1165 PRINT E; "POWER, THE ANSWER IS"
1170 FOR N=1 TO P
1180 CURSOR 8, N
1190 IF N>60 THEN CURSOR 9, N-60
1200 LET Z=A2(N)+48
1210 SET DB=Z
1220 NEXT N
1250 PRINT
1260 GOTO 1320
1270 PRINT "NUMBER TOO BIG, OVERFLOW"
1280 GOTO 1320
1290 PRINT "ZERO TO ANY POWER IS ZERO"
1300 GOTO 1320
1310 PRINT "ANY NUMBER TO THE ZERO POWER IS ONE"
1320 PRINT "TYPE 1 FOR MORE, 0 TO STOP"
```

```
1330 INPUT S
1340 IF S=1 THEN GOTO 90
1350 END
```

Program IPOWR has been nicely documented—nearly every operation is prefaced by a short explanatory statement, and nearly every variable is defined. Still, the mechanics of IPOWR are not easily understood without study.

CONDENSED CODING

There are numerous loops and loops-within-loops, and condensed coding such as the following is common:

```
870 LET T2=(T1-10*(INT(T1/10)))+B2(D)
```

The remarks at the beginning of IPOWR state that the program "finds the value of *A* to the *B* power where *A* and *B* are both integers and the result is accurate to 100 places."

What is so special about that? Raising one number to the power of another number is no great task; we merely need write:

```
LET C=A^B
```

Suppose we were to do precisely that. Suppose further that we were to choose:

```
A=14
B=23
```

The result would be printed out:

```
C=2.2958579E+26
```

The SOL BASIC compiler will let us specify output formats as either integer, floating-point, or exponential notation. If we want greater accuracy, we might try to print the result out in integer format as in the following example. After all, a whole number multiplied by itself *n* times will still yield a whole number.

```
PRINT %26I;C
```

Unfortunately, the largest field we may specify on the SOL (and this is similar to most other machines) is a field 26 digits wide. The result of our effort in the above example, then, is:

```
FO ERROR IN LINE_____
```

The term *FO* means *field overflow;* we have tried to display a number larger than the specified format will allow. We should have seen that the number would not fit a 26I field: *E*+26 implies 26 digits to the right of the decimal point, not including the digit 2 on the left of the decimal point. The upper limit on field size is 26 digits only, not 27, so our number is too large for the format.

What if we were to try an exponential representation, with the maximum number of decimal places displayed? Again, the upper limit on total number of characters is 26, and after taking into account plus and minus signs and room for the exponent, the maximum number of digits to the right of the decimal point is 19, as follows:

```
PRINT %26E19;C
C=2.29585790000000000000E+26
```

It does not look as though we gained any accuracy: we now have the number represented by 20 digits, but 12 of those digits are zeros. If we do some long multiplication by hand, we discover that those trailing zeros are purely an artifact of the way our computer handles its internal representation of numbers.

As an example, raising 14 only to the eighth power instead of the twenty-third as in previous examples, long multiplication results in:

$$14^8 = 1475789056$$

Already we have two digits of greater accuracy than the computer can provide. The computer's problem is that it cannot keep track of more than 8 digits at a time. What we need (if we are to have answers accurate to more than 8 digits) is a program that mimics hand multiplication, a program which (1) takes first the *units* digit of the multiplier and multiplies it by all of the digits in the multiplicand, then (2) takes the *tens* digit and does the same, then (3) moves on to the *hundreds* digit, (4) the *thou-*

sands digit, and so on as required, until all digits of the multiplier have been multiplied against all digits of the multiplicand, and the many intermediate results have been added up to produce the final result:

```
  4572
× 326
------
 27432 <-------- result of 4572 × 6
 9144  <-------- result of 4572 × 2
13716  <-------- result of 4572 × 3
-------
1490472 <------- total
```

Apparently, IPOWR is supposed to do just that: digit-by-digit multiplications between numbers to arrive at an answer accurate to 100 places. Considering that the unaided computer is accurate only to 8 places, the program IPOWR, if it works, represents a considerable improvement.

In order to test IPOWR, we must first provide an accurate standard of reference. We can do this by performing a hand multiplication on some number raised to a conveniently low power. For this first trial, the answer need not exceed 8 digits, although that aspect of the program will have to be tested eventually.

Suppose we try 5 × 5. We write:

```
  5
× 5
---
 25
```

Using the computer, we would have:

```
? 5, 2
WORKING. . .
FOR THE NUMBER 5 TO THE 2 POWER, THE
ANSWER IS 25
```

No problem there. Let's try a two-digit number, something easy:

```
  10
× 10
----
 100
```

and, using IPOWR:

```
? 10, 2
WORKING . . .
FOR THE NUMBER 10 TO THE 2 POWER, THE
ANSWER IS 100
```

Everything appears to be working correctly, but 10 is a special-case number, one which does not test those parts of the program involved with carrying over partial results, etc. We decide to choose a more representative number:

```
   46
 × 46
 ----
  276
 184
 ----
 2116
```

and:

```
? 46, 2
WORKING . . .
FOR THE NUMBER 46 TO THE 2 POWER, THE
ANSWER IS 1916
```

A bug! The answer is incorrect by 200. We try another set of values, first by hand:

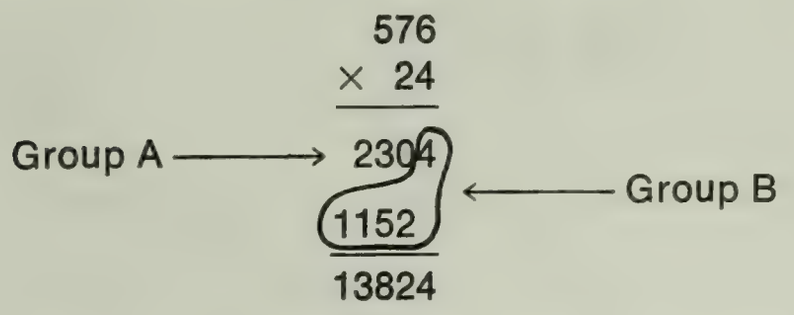
```
   24
 × 24
 ----
   96
  48
 ----
  576
 × 24
 ----
 2304
1152
----
13824
```

then using the program:

```
? 24, 3
WORKING . . .
```

```
FOR THE NUMBER 24 TO THE 3 POWER, THE
ANSWER IS 11524
```

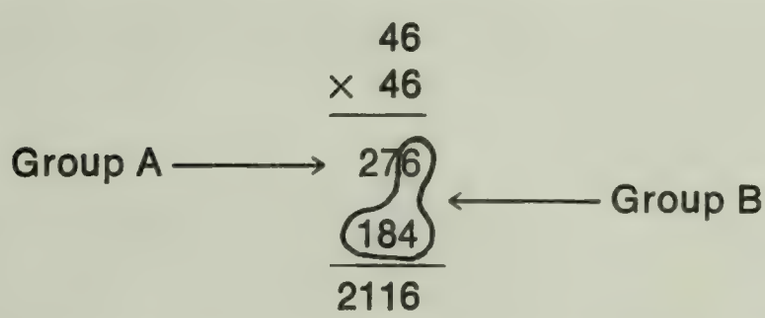
This time the computer's answer is off by 2300. The magnitude of the two errors can perhaps give us some clue as to the origin of the bug, but 2300 does not seem related in any simple way to the first error we observed of 200. If we study the second level of the hand multiplication, however, something very interesting presents itself.



If we take the digits circled in group B, we have 11524, the same as the incorrect value given by the computer. In addition, the digits in group A (after allowing for the digit 4 which was used to occupy the *units* place) equal 2300, the magnitude of the error.

It would seem we have found the bug. IPOWR is not correctly summing the temporary results. Is this true? If it is, we should observe the same bug for all other calculations involving two intermediate sums.

We can check our hypothesis by looking at the previous example:



Applying the same logic to this example, we could predict that the answer would be 1846, with a resulting error of 270. As we saw, however, in this case the program gave an answer of 1916, and an error of 200. Our method does not work in this case. Consequently, we must say that our explanation for the bug, while it looks good for the case of 24^3, did so only by a fortunate set of circumstances, and no more.

We are going to have to look much deeper to find the cause of this bug.

IPOWR is much longer than any of the programs we've looked at so far; but by using techniques already familiar to us we can break the program down into workable blocks.

LINE-BY-LINE CHECKING

We begin with the flowchart representation of the IPOWR initialization sequence (Fig. 2-2). The flowchart will give us the structural outline of this phase of IPOWR, while the program listing will tell us precisely what is taking place.

We are now playing computer. We will do whatever we are commanded to do by the lines of code. As we execute the program we will use a pad of paper and a pencil as our memory. In the interest of saving time, however, we will not find it necessary to list the entire contents of a 100-element array, for instance.

The first thing we must do is allocate room in our memory for five different arrays, as in Fig. 2-3. We cannot correctly say that the arrays, since they are newly created, contain nothing, and that "nothing" is logically and mathematically equivalent to zero. If we want each memory location of each array to contain a zero, we must actively load

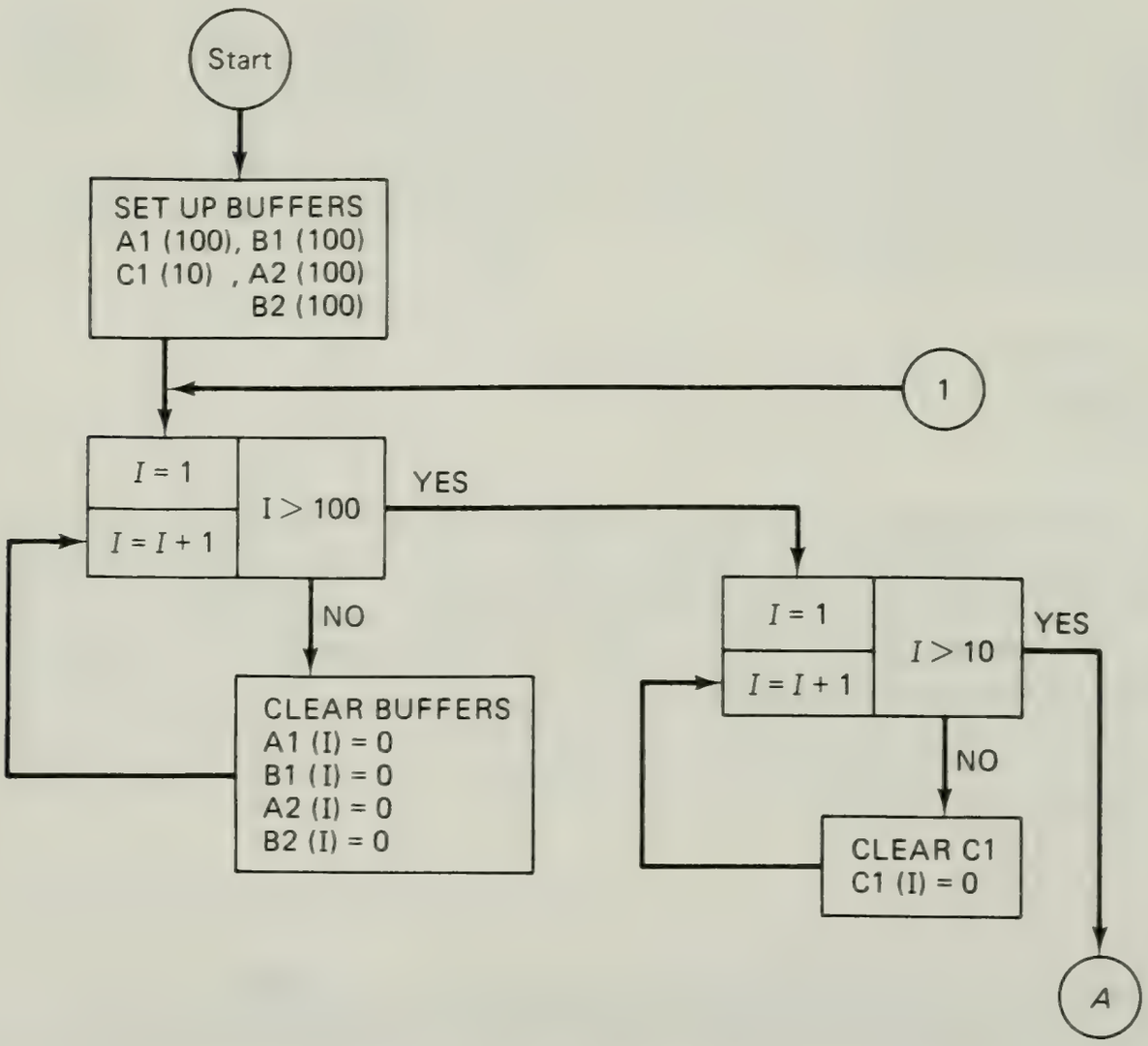


FIG. 2-2. Flowchart, program IPOWR—initialization and clearing of the buffers.

FIG. 2-3. Representation of the arrays in memory, after creation by DIM statement.

it with a zero. After we were to so load array A1 would look like Fig. 2-4. Arrays A2, B1, B2, and C1 would be similar.

Now we can proceed to the next flowchart segment, illustrated in Fig. 2-5. (Remember, these segments are taken from the master flowchart of Fig. 2-1.) We are now told to get the variables *R* and *E* from the keyboard. We already know that 46^2 gave an incorrect answer of 1916 when it should have been 2216, so since there is only one level of multiplication involved (simplifying our manual trace), we decide to use those same numbers:

R=46
E=2

Exponentiating a number, or raising a number to the power *n*, means that we will multiply that number times itself *n*−1 times. We will therefore be using *E* as an index variable for a master loop over the total number of multiplications. But in order to do so correctly we must first decrease the value of *E* by 1:

E=E−1=2−1=1

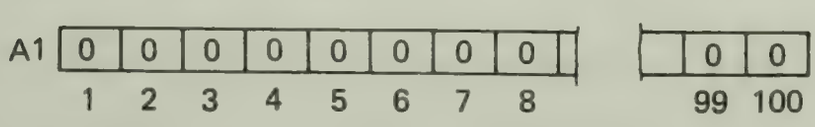


FIG. 2-4. Array A1 cleared and initialized with all zeros. Arrays A2, B1, B2, and C1 would look similar.

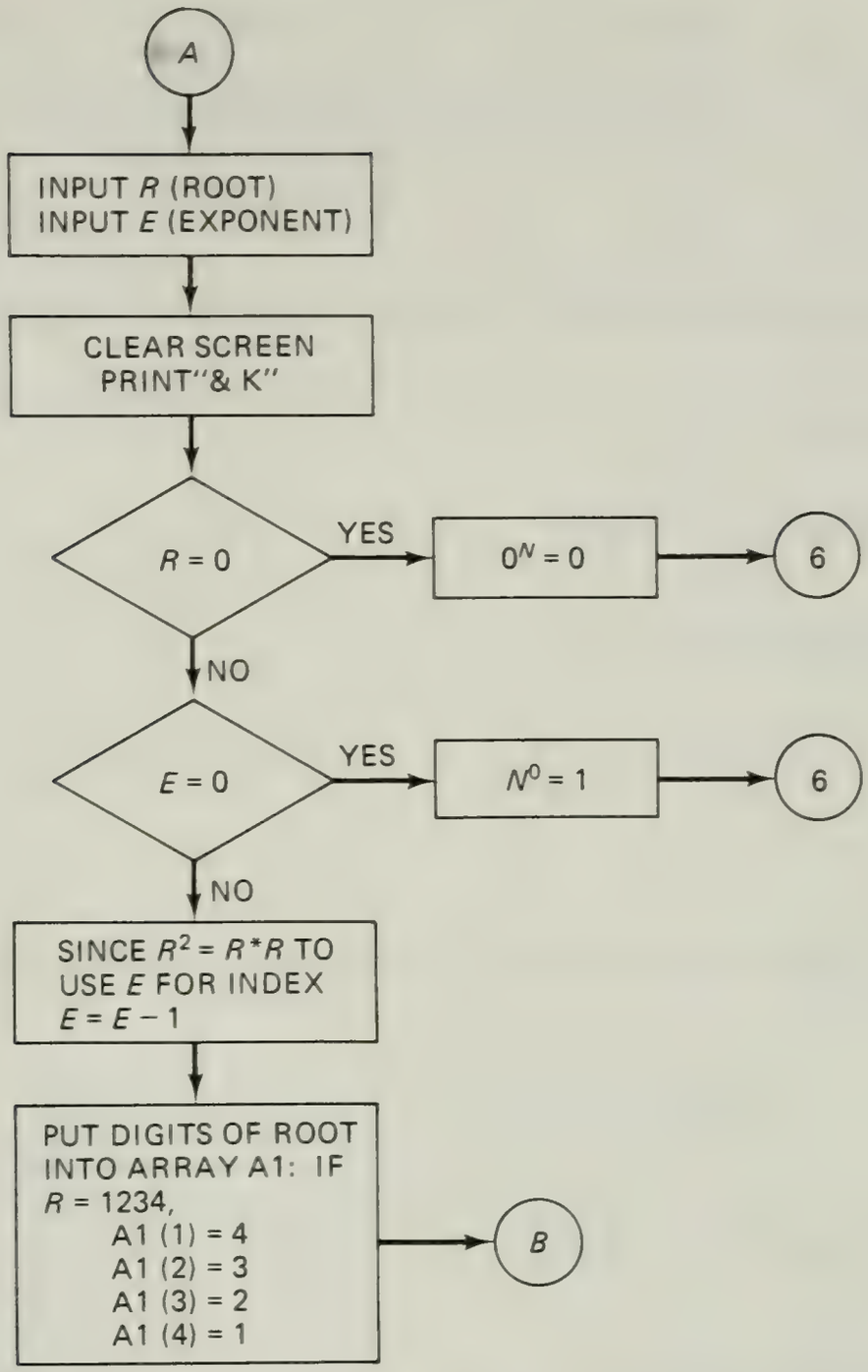


FIG. 2-5. Flowchart, program IPOWR—input of variables, test for zero values, load root array.

Since we will be performing a digit-by-digit multiplication we must break up the root value *R*, one digit at a time, and store those digits in array A1. The algorithm (the pattern) we use to do this is:

```
480 LET A1(J)=R-(10*INT(R/10))
```

The computer's operational hierarchy begins with the innermost set of parenthesis and works out, each time performing mathematical operations according to the rules:

Priority	*Operation*
1	- (unary negate)
2	^ (exponentiate)
3	* , / (multiply, divide)
4	+ , − (add, subtract)

We, as computer, must do the same. Using $R = 46$, we would have:

```
1. (R/10) = 46/10
          = 4.6
2. INT(R/10) = INT(4.6)
             = 4
3. 10*INT(R/10) = 10*4
                = 40
4. R-(10*INT(R/10)) = 46-40
                    = 6
A1(1) = 6
```

The first time through the loop J would equal 1, so array A1 would look like Fig. 2-6.

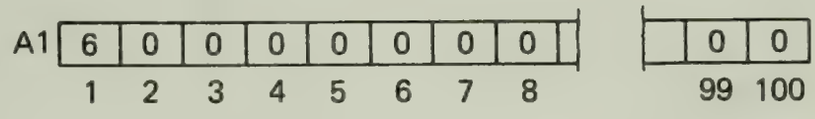


FIG. 2-6. Array A1 with units digit of the multiplicand loaded into the first element.

The next command:

```
490 LET R =INT(R/10)
```

replaces R with a new value. The phrase on the right side of the equal sign looks familiar from before. R still equals 46 going into this line of code:

```
R=46
1.  R/10 = 46/10
         = 4.6
2.  INT(R/10) = INT (4.6)
              = 4
```

3. R=4

We now loop back up to line 450 where we increment the value of *J*:

```
J=J+1
 =1+1
 =2
```

and begin again. This time:

```
R=4
J=2
 A1(J)=R-(10*INT(R/10))
1.  R/10 = 4/10
         = .4
2.  INT (R/10) = INT(.4)
               = 0
3.  10*INT(R/10) = 10*0
                 = 0
4.  R-10*INT(R/10) =4-0
                   = 4
A1(2) = 4
```

Array A1 now looks like Fig. 2-7.

Again we find a new value for *R*:

```
R=INT(R/10)
  =INT(4/10)
  =0
```

Since *R*=0, the test at line 510 fails:

```
510 IF R>0 THEN GOTO 450
```

and we proceed in the execution of the program.

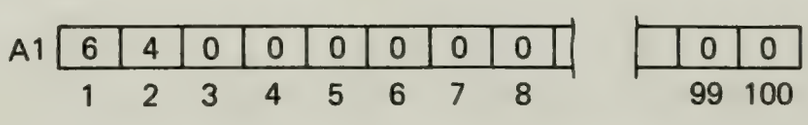


FIG. 2-7. Array A1 with both digits of the multiplicand loaded into the first two elements.

Since we want to multiply *R* times itself *E* number of times, we must first set the multiplier and the multiplicand equal to each other. We do this with a simple loop, after which arrays A1 (multiplicand) and C1 (multiplier) look like Fig. 2-8.

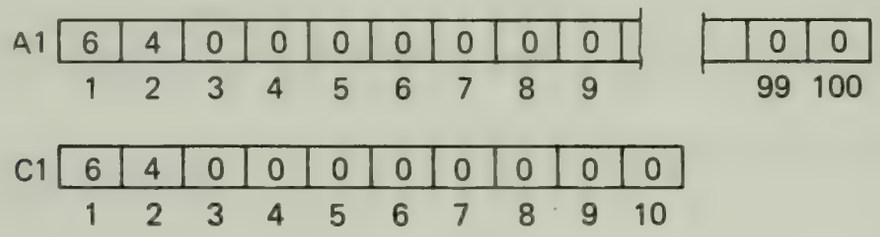


FIG. 2-8. Equating of arrays A1 (multiplicand) and C1 (multiplier).

We now begin the master loop over the number of multiplications, indexed by *E*:

```
570 FOR I=1 TO E
```

CHECKING INDEX VARIABLES

Indexing is always a good source of bugs, so we will want to pay close attention to how the index values are used and updated. To make the task a little easier we will start a special table where we can keep track of index variable values, as in Fig. 2-9. We're working in that segment of the flowchart depicted in Fig. 2-10.

IPOWR spends the bulk of its time looping over 100-element arrays like A1, A2, etc. We can speed up program execution if we only concern ourselves with those elements which are active. One way to do this is to do a floating-point multiplication operation and find out from it how many active digits are required to represent the answer.

Index	Function	Value
I	EXPONENT	1

FIG. 2-9. Index variable status for first loop, at line 570.

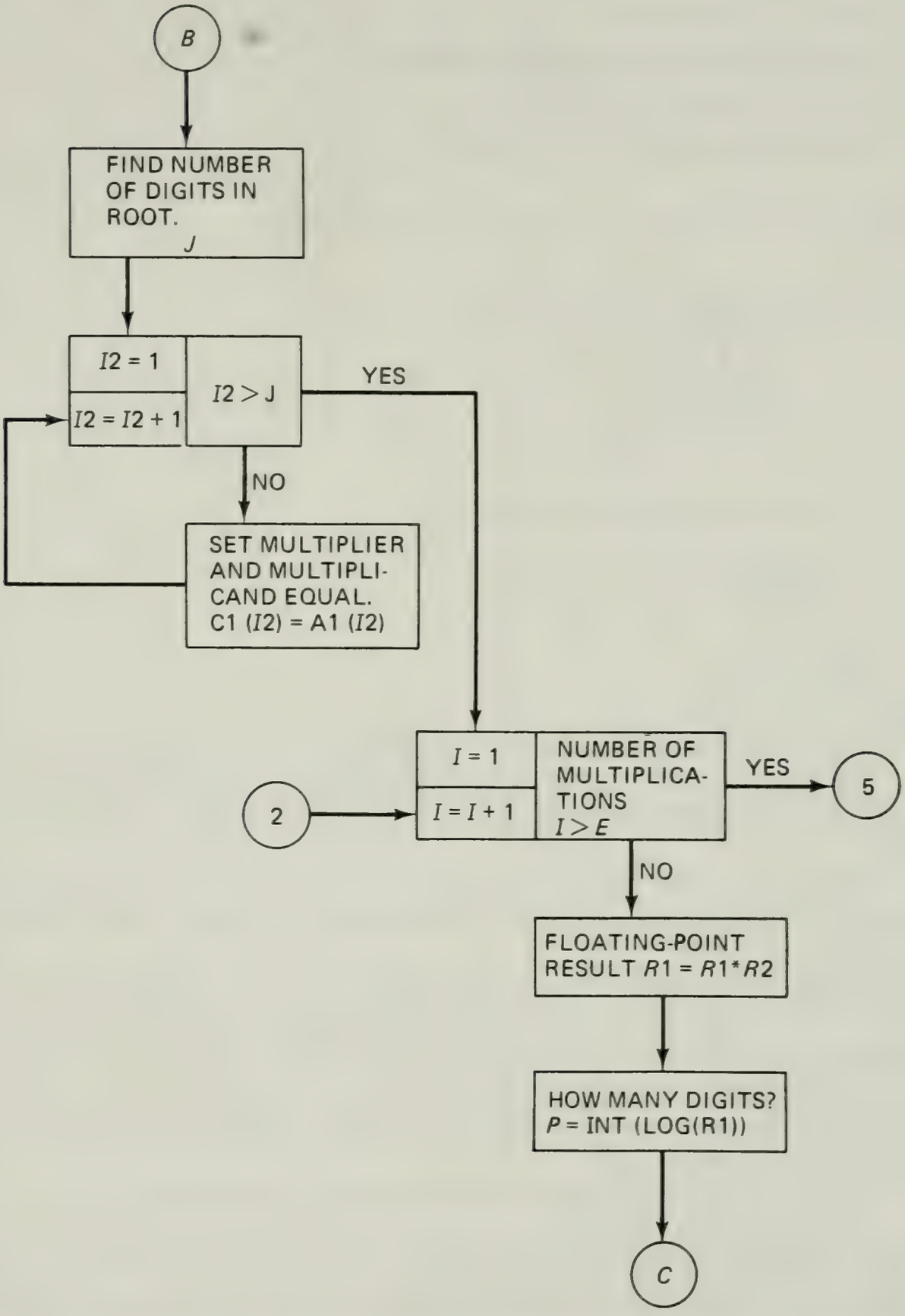


FIG. 2-10. Flowchart, program IPOWR—equate multiplier and multiplicand, begin loop over total number of multiplications, and calculate number of digits in current temporary result.

```
590 LET R1=R1*R2
600 REM HOW MANY DIGITS IN RESULT?
610 LET P=INT(LOG10(R1))+1
620 REM P=NUMBER OF PLACES
```

In our role as computer, we would do the following:

1. R=R1*R2
   =46 × 46
   =2116.0
2. LOG10(R1)=LOG(2116)
   =3.3255
3. INT(LOG10(R1))=INT(3.3255)
   =3
4. INT(LOG10(R1))+1=3+1
   =4

P=4

There are in fact four digits which make up the number 2116.

We now enter the many loops-within-loops of program IPOWR. A general outline of the program's structure in this regard is shown in Fig. 2-11.

The asterisk in Fig. 2-11 identifies our current position, while the loops are all marked with their function and the name of their index variable.

We note that loop 2 clears the temporary result storage array, B2. This array was already cleared and loaded with zeros at the top of the program, but we do it again here. Why? Because after every transit through the main loop, the temporary array will contain data from the previous calculation, which must be cleared away if the current calculation is going to be correct.

We use loop 4 to clear the carry array for the same reason.

Arriving at line 790, we are inside the innermost loop and at the meat of the calculation. We check the status of our index variables in Fig. 2-12, and we execute line 790. The flowchart segment now being processed is illustrated in Fig. 2-13. Our index table gives us the current values of M1 and M2, and we can check our arrays to see what values are being pointed to. We find:

```
A1(M2)=A1(1)=6
C1(M1)=C1(1)=6
B1(M2)=B1(1)=0
```

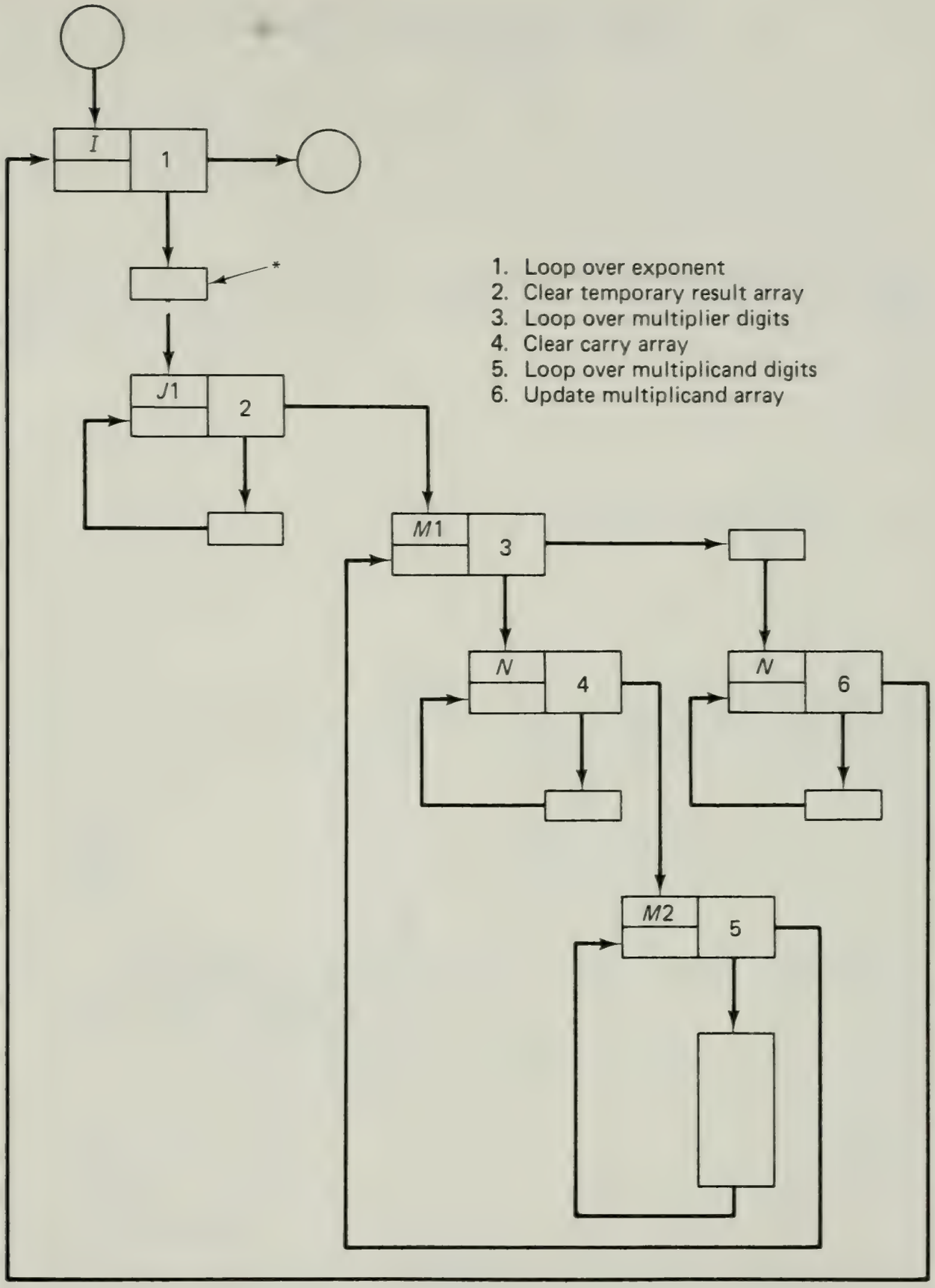


FIG. 2-11. Flowchart of program IPOWR showing those loops directly concerned with digit-by-digit multiplication.

Index	Function	Value
I	EXPONENT	1
*M*1	MULTIPLIER	1
*M*2	MULTIPLICAND	1

FIG. 2-12. Index variable status for first loop, at line 790.

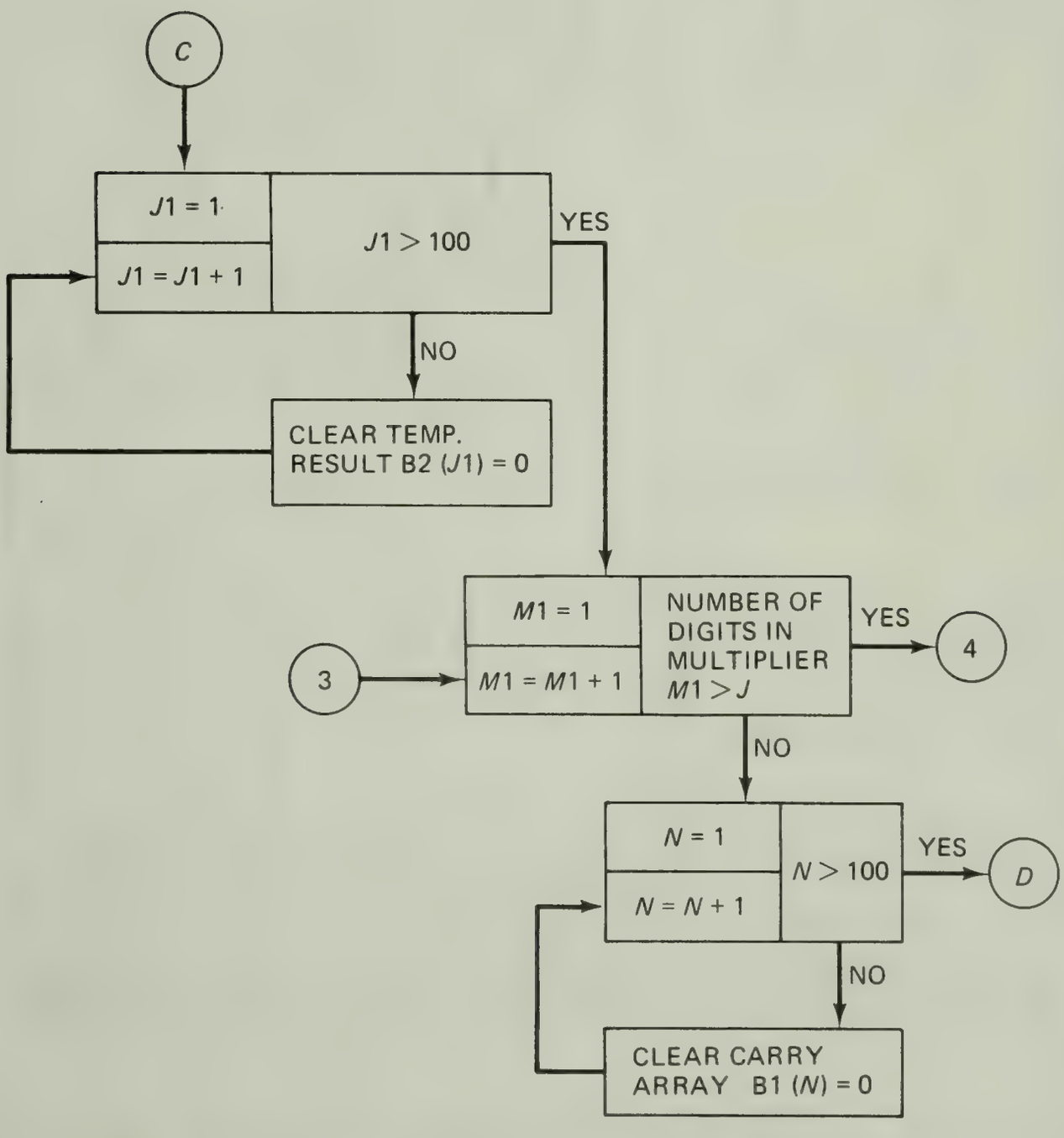


FIG. 2-13. Flowchart, program IPOWR—temporarily clear result buffer, begin loop on number of digits in multiplier, and clear carry array.

so we may decode line 790 to read as follows:

```
LET T1=A1(1)*C1(1)+B1(1)
    T1=6*6+0
    T1=36
```

It might help if we had stopped to relate these events to the steps which one must go through to accomplish multiplication by hand. Our object is to multiply 46 by 46:

```
  46
× 46
----
```

We may identify the two participants in the multiplication as follows:

```
  46 multiplicand
× 46 multiplier
----
```

Perhaps now it is easier to understand why we loaded the arrays A1 and C1 in apparent reverse order; in multiplication we first multiply the units digit of the multiplier times the units digit of the multiplicand:

```
  46
× 46
----
  36
```

But the 3 of 36 does not go below the line; it is a carry-over number and is written above the multiplicand as follows:

```
   3
  46
× 46
----
   6
```

Program IPOWR accomplishes this by first determining where to place the units digit of the temporary result (Fig. 2-14).

```
810 REM WHAT IS OUR DIGIT POSITION?
820 LET D=M1-1+M2
```

Referring back to our table of index values, we know that:

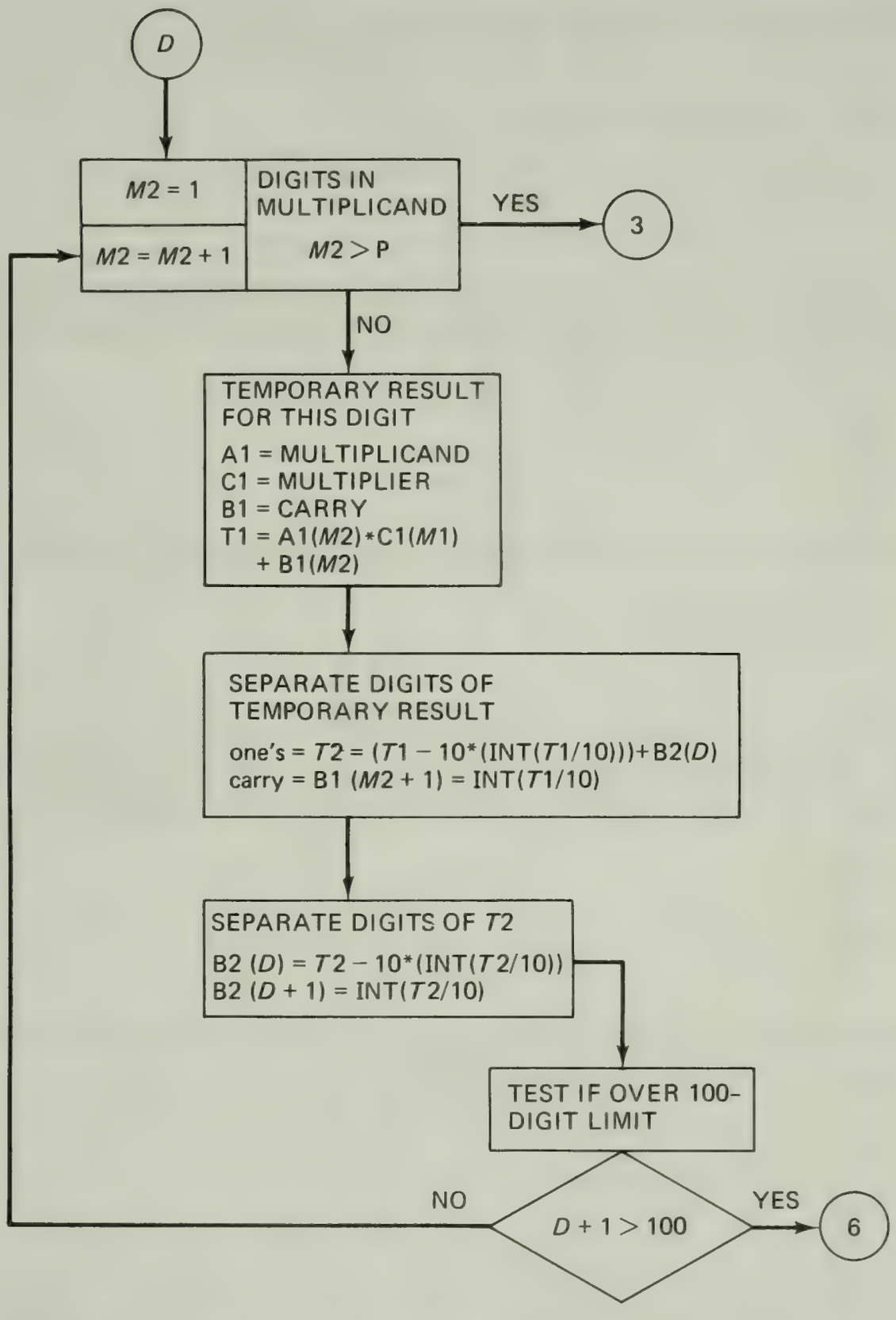


FIG. 2-14. Flowchart, program IPOWR—loop over digits in multiplicand, find temporary results for each digit, add the units, carry the tens, and check for overflow.

$M1 = 1$

$M2 = 1$

therefore

$$D = M1 - 1 + M2$$
$$= 1 - 1 + 1$$
$$= 1$$
$$D = 1$$

The 6 of 36 therefore belongs in the first digit place. We cannot, however, simply assign it to that space and continue. This same block of code must be correct for any number in any position, which means that there may be occasions in which the units digit of the just-calculated temporary result must be added to a digit already occupying that space to yield the final result. Line 870 does this:

```
870 LET T2=(T1-10*(INT(T1/10)))+B2(D)
```

We know that *D* is the pointer to the current digit position, and the coding underlined is also familiar to us as a pattern which replaces the rightmost digit of a number with zero. We have, therefore:

$$T2 = 36 - 30 + B2(1)$$
$$= 6 + B2(1)$$
$$= 6 + 0$$
$$= 6$$
$$T2 = 6$$

The quantity *B2(1)* is the digit from a previous calculation referred to earlier. But why couldn't we have performed the following?

```
B2(D)=(T1-10*(INT(T1/10)))+B2(D)
```

The reason we cannot perform the above calculation is that T2 might very well be a two-digit number, and each of the arrays—A1, A2, B1, B2, and C1—may only hold one digit per element. Lines 900 and 920 take care of this task by breaking T2 up into its two component digits and assigning them to neighboring positions in array B2:

```
900 LET B2(D)=T2-10*(INT(T2/10))
920 LET B2(D+1)=INT(T2/10)
```

Updating the carry array, B1, is accomplished at line 960:

```
960 LET B1(M2+1)=INT(T1/10)
```

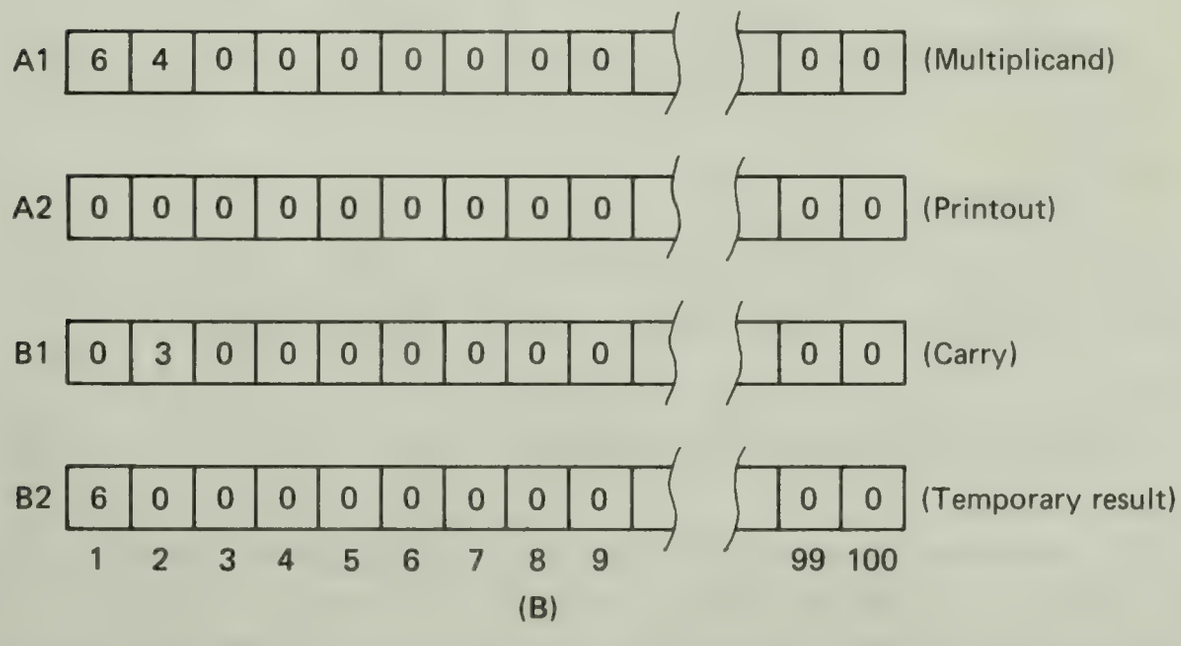
Now is a good time to review the index values and the contents of the various arrays, as in Fig. 2-15.

If we were to lay a grid on top of our hand calculation, and label it to correspond to the various arrays being used in IPOWR, we would find results as in Fig. 2-16. The similarity to our hand method is obvious.

We have finished our first traverse through loop 5, so we must now go back up to statement 740, increment the index variable M2, and go through the loop again.

(A)

Index	Function	Value
I	EXPONENT	1
*M*1	MULTIPLIER	1
*M*2	MULTIPLICAND	1
D	DIGIT POINTER	1



(B)

FIG. 2-15. In (A), index variable status (first traverse through each loop, at line 900); (B) shows array contents at end of first digit multiplication. Note that the 3 was carried into element B1(2).

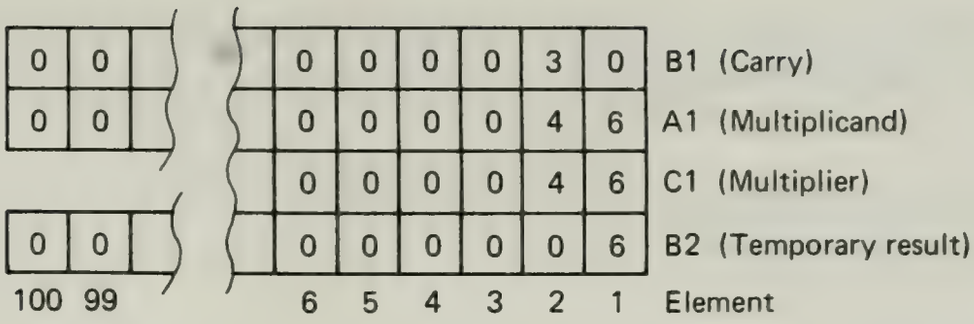


FIG. 2-16. Array grid overlaid on conventional representation of hand multiplication to show relationship to various elements.

As before, we execute line 790:

```
790 LET T1=A1(M2)*C1(M1)+B1(M2)
```

The index values are found in Fig. 2-17 so we interpret line 790 to be:

```
T1=A1(2)*C1(1)+B1(2)
  =4*6+3
  =24+3
  =27
T1=27
```

We find an updated digit position pointer:

```
820 LET D=M1-1+M2
        D=1-1+2
        D=2
```

Index	Function	Value
I	EXPONENT	1
*M*1	MULTIPLIER	1
*M*2	MULTIPLICAND	2

FIG. 2-17. Index value status—units digit of multiplier times tens digit of multiplicand, at line 790.

And as before, the units digit (7) is chopped off of the temporary result T1, and added to the digit already occupying that position:

```
870 LET T2=(T1-10*(INT(T1/10)))+B2(D)
        T2=7+0
          =7
```

In order to be general we must allow for the possibility that T2 is a two-digit number (although in this case it is only one digit long). We go through the coding at lines 900 and 920 to split T2 into its units-digit and tens-digit components, and assign them respectively to spaces *B2(D)* and *B2(D+1)* of the temporary result array.

Also as before, the tens digit of the first temporary result, T1, is chopped off and installed as the new carry digit in array B1:

```
950 REM FIND NEW CARRY DIGIT
960 LET B1(M2+1)=INT(T1/10)
```

Using the display method of Fig. 2-16, we may check our progress as in Fig. 2-18.

We know that loop 5 (Fig. 2-11) is set up to multiply each digit from array C1 (as currently specified by the index variable M1 from loop 3) times a total of *P* number of elements from array A1:

```
740 FOR M2=1 TO P
```

Loop 5 could have been indexed from 1 to 100, which would cause each digit of the multiplier to operate on all 100 elements of the multiplicand array, regardless of whether or not all of those elements were im-

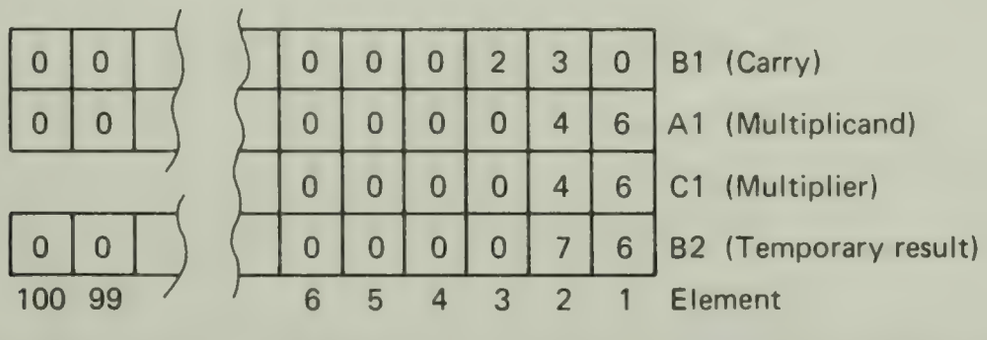


FIG. 2-18. Array contents—unit digit of multiplier times tens digit of multiplicand, showing updating of array B2 and digit carry in array B1.

portant. By finding out beforehand how many digits comprise the answer, as we did when we calculated *P*, we save ourselves the considerable extra time involved in looping through all 100 elements, while still insuring that all required digits are operated on.

Loop 5 therefore executes four times in this example, since *P*=4. Our index array tables now have the appearance of the elements shown in Fig. 2-19.

We can check that 46 × 6 = 276, so this part of the program works correctly. We have not yet found the bug, but if we continue to follow and execute the program just as the computer must, we will eventually find it.

Having completed our final transit through loop 5 we find ourselves also at the end of loop 3:

```
970 NEXT M2
980 NEXT M1
```

We now go back to line 680 where we increment M1, the loop 3 index, cycle through loop 4 to clear the carry array B1, then increment the loop 5 index M2, in order to begin the this-time multiplication of the multiplicand by the *tens* digit of the multiplier.

Index	Function	Value
I	EXPONENT	1
*M*1	MULTIPLIER	1
*M*2	MULTIPLICAND	4

(A)

0	0			0	0	0	2	3	0	B1 (Carry)
0	0			0	0	0	0	4	6	A1 (Multiplicand)
				0	0	0	0	4	6	C1 (Multiplier)
0	0			0	0	0	2	7	6	B2 (Temporary result)
100	99			6	5	4	3	2	1	Element

(B)

FIG. 2-19. In (A), index variable status (units digit of multiplier cycling through first four elements of multiplicand, as required by variable *P*); (B) shows array status at the end of units-digit multiplication—the intermediate result.

We check our index and array status, as in Fig. 2-20. We may list the contents of loop 5, without the intervening comments, in only a few lines:

```
LET T1=A1(M2)*C1(M1)+B1(M2)
LET D=M1-1+M2
LET T2=(T1-10*(INT(T1/10)))+B2(D)
LET B2(D)=T2-10*(INT(T2/10))
LET B2(D+1)=INT(T2/10)
LET B1(M2+1)=INT(T1/10)
```

Since we are playing computer, we must supply the proper index values and do the computations ourselves:

```
T1=A1(1)*C1(2)+B1(1)
  =6*4+0
T1=24

D=M1-1+M2
 =2-1+1
D=2
```

Index	Function	Value
I	EXPONENT	1
*M*1	MULTIPLIER	2
*M*2	MULTIPLICAND	1

(A)

100	99		6	5	4	3	2	1	Element
0	0		0	0	0	0	0	0	B1 (Carry)
0	0		0	0	0	0	4	6	A1 (Multiplicand)
			0	0	0	0	4	6	C1 (Multiplier)
0	0		0	0	0	2	7	6	B2 (Temporary result)

(B)

FIG. 2-20. In (A), index variable status at beginning of loop over second digit of multiplier; (B) is array status, with cleared carry array in preparation of cycle over tens digit of multiplier.

```
T2=(TI-10*(INT(T1/10)))+B2(D)
  =(24-10*(INT(24/10)))+7
  =(24-20)+7
  =4+7
T2=11

B2(D)=T2-10*(INT(T2/10))
     =(11-10)
B2(2)=1

B2(D+1)=INT(T2/10)
       =INT(11/10)
B2(3)  =1

B1(M2+1)=INT(T1/10)
        =INT(24/10)
B1(2)   =2
```

We may now use these values to update our arrays, as in Fig. 2-21. (The flowchart segment is shown in Fig. 2-22.)

But something is obviously incorrect. This new temporary result is less than the previous temporary result. We have in all probability found the bug. The problem now is to analyze the symptoms and, if possible, determine the cause.

The final expression, *B2(1)*, equals 6, as it did at the end of the first pass through loop 3. This is as it should be, since *D* pointed to the second element. The program correctly multiplied 4 times 6 to give 24; it carried the 2, added the 4 to the 7 already present at location *B2(2)*, to yield 11, and then it placed the units digit in *B2(2)* and the tens digit in *B2(3)*.

But what about the 2 already occupying *B2(3)*? Instead of replacing

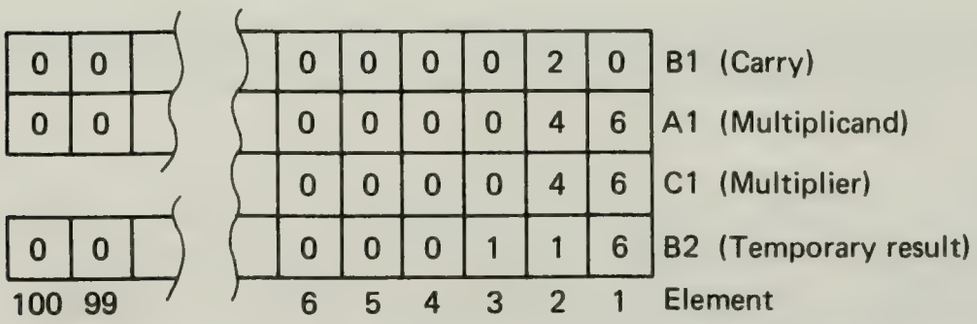


FIG. 2-21. Array contents—tens digit of multiplier times units digit of multiplicand. Note the incorrect temporary sum in array B2.

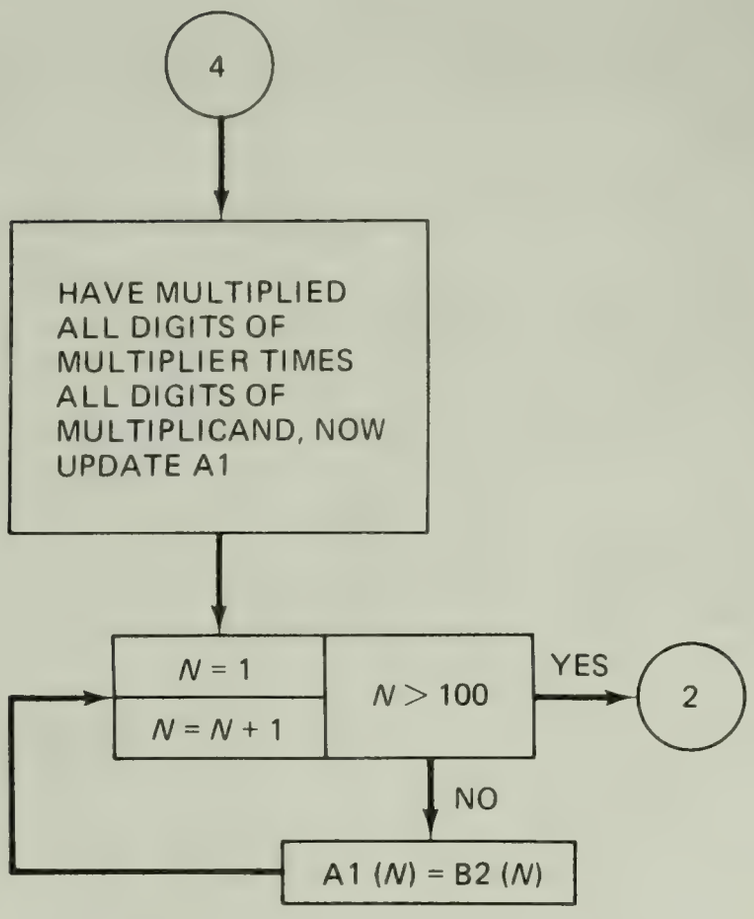


FIG. 2-22. Flowchart, program IPOWR—exit from main loop, at which point all digits of multiplier have been multiplied by all digits of multiplicand. Update array A1 with results of this multiplication to prepare for next cycle.

the 2 of *B2*(3) with a 1 from the tens digit of 11, we should instead have added 2 and 1 to give 3.

The offending line of code is 920:

```
920 LET B2(D+1)=INT(T2/10)
```

It should be modified as follows:

```
920 LET B2(D+1)=INT(T2/10)+B2(D+1)
```

We can easily use the SOL's editing capability to modify line 920, and then rerun the program to see if we did the right thing:

```
RUN
? 46,2
WORKING . . .
FOR THE NUMBER 46 TO THE 2 POWER, THE
ANSWER IS 2116
```

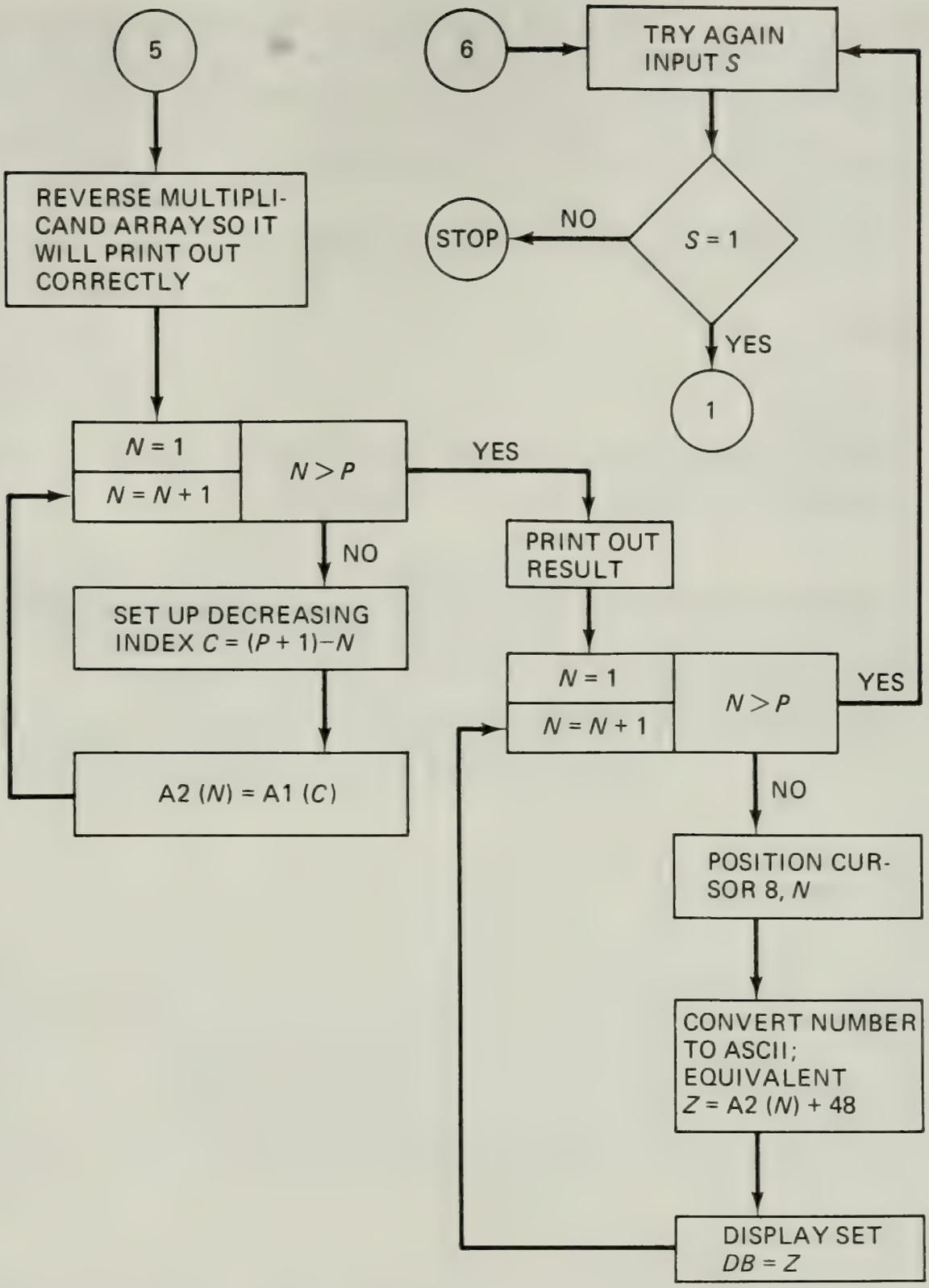


FIG. 2-23. Flowchart, program IPOWR—have finished R^E, transfer final answer to print array in reverse order. Display result one digit at a time. Then test if program is to be repeated. If not, stop.

We could continue to trace through the balance of the program (Fig. 2-23) on paper, playing computer as we have been doing, for an exercise. But we seem to have found and corrected the bug so it is not necessary that we do so.

Since the SOL is only accurate to eight digits, a restraint shared by

other easily available tools such as pocket calculators, there really is no way to check the upper-accuracy limit of IPOWR save through long and involved hand calculations. But by using examples that fall within those limits, it is possible to test IPOWR and convince ourselves that the bug has indeed been stamped out and that IPOWR is now a correctly functioning program.

Trying IPOWR again on the original example of 14^{23}, we have:

```
RUN
? 14, 23
WORKING . . .
FOR THE NUMBER 14 TO THE 23 POWER, THE
ANSWER IS 2291807325125827717292605 44
```

The method of debugging we used for this program is long and involved, but it is thorough. By playing computer we put the program under a microscope, examining each step and calculation in detail. And bugs, once discovered, are then obvious and easily fixed.

3

Debugging with Print Statements

Print statements are among the easiest to use and the most often used debugging tools available to the computer programmer: They are straightforward to construct, are simple to insert into existing blocks of code, and have the potential of yielding large amounts of information. The challenge, when using print statements to debug a program is in determining which variable quantities to display and at what point in the program to do so.

INSERTING PRINT STATEMENTS

Consider the following example: The specifications call for a program that will sort, in descending order, an arbitrarily long list of numbers and then display the result. The value -1 will signal that the last number has been input and that the sorting process is to begin.

Suppose we generate the flowchart shown in Fig. 3-1 to describe the sorting algorithm. Tracing through the diagram we can see that the first step is to set up storage space for two arrays, A and B. We then initialize an index variable *I* and read in our first value, storing it in the

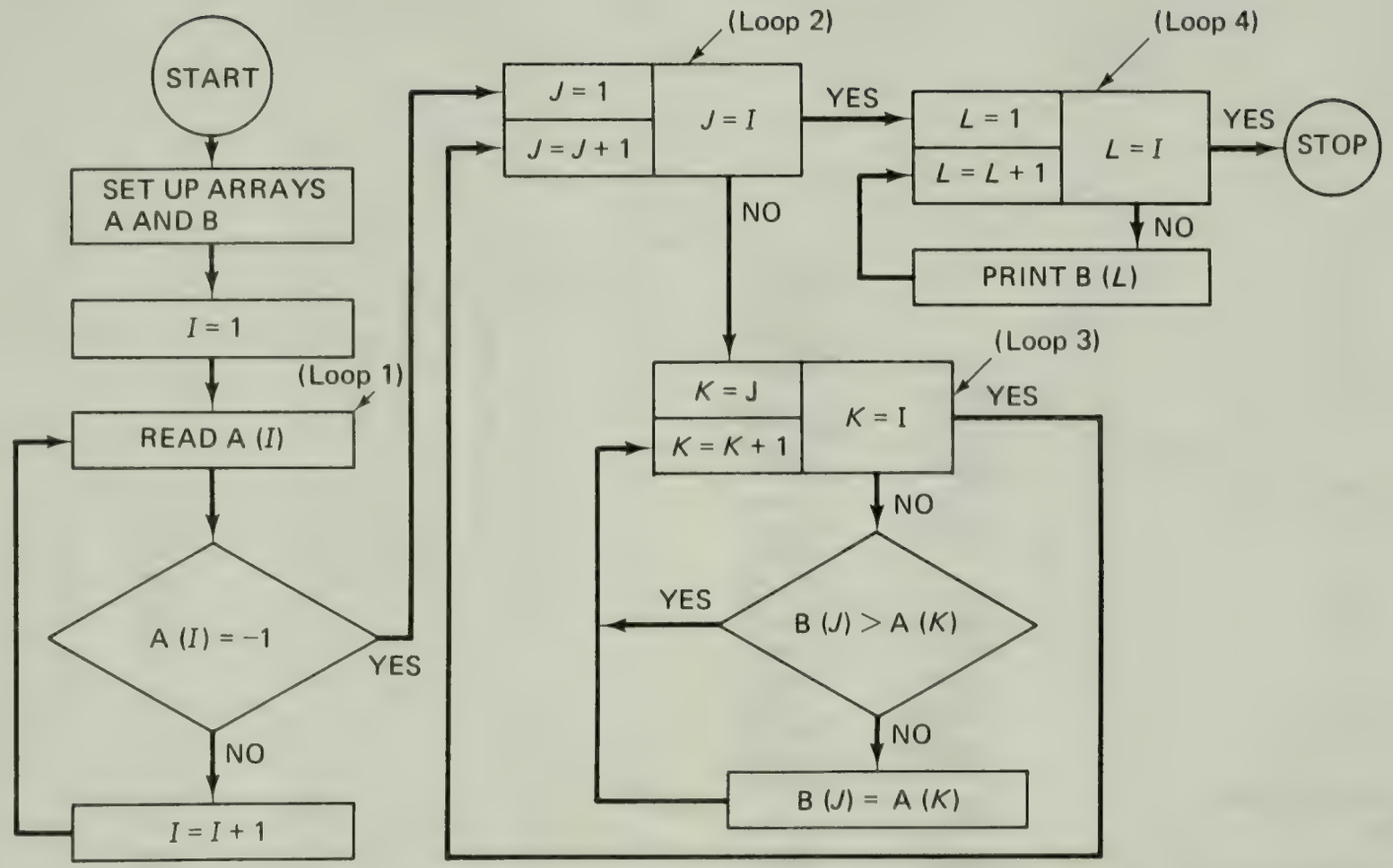


FIG. 3-1. The original flowchart for the number-sorting routine.

first element of array A. If the value just read in equals -1, the termination value, we branch over to the sorting routine. If A(*I*) does not equal -1, we increment the index *I* and read in another variable.

Once all of the values have been accumulated, we enter the actual number-sorting phase of the program. Two concentric loops perform the sorting operation; the outer loop indexes the B array for the result, and the inner loop indexes the A array, which contains the unsorted data. If the current value from the A array is greater than that from the B array against which it is being compared, the B array element is set equal to it. If the A value is less than the B value, the next A value is fetched and the testing begins again.

When all of the comparisons are done, we branch to a final loop to print out the answer as it appears in array B.

The program which results from the flowchart is:

```
5 REM PROGRAM TO SORT NUMBERS
10 REM SET UP ARRAYS
20 DIM A(100), B(100)
30 PRINT "YOU MAY ENTER UP TO 100 NOS, -1 ENDS LIST"
40 REM INITIALIZE INDEX
50 LET I=1
60 REM READ DATA, STOP IF = -1
70 INPUT A(I)
80 IF A(I)=-1 THEN GOTO 110
90 LET I=I+1
100 GOTO 70
110 REM SORT LIST OF NUMBERS
120 FOR J=1 TO I
130 FOR K=J TO I
140 IF B(J)>=A(K) THEN GOTO 170
150 LET B(J)=A(K)
160 GOTO 130
170 NEXT K
180 NEXT J
190 REM PRINT OUT ANSWER
200 FOR L=1 TO I
210 PRINT B(L)
220 NEXT L
230 END
```

Apparently the largest numbers are sorted out first, so in order to keep a test run as simple as possible it would be permissible to use a sequence of values such as 1, 2, and 3, with results as follows:

```
RUN
YOU MAY ENTER UP TO 100 NOS, -1 ENDS LIST
? 1
? 2
? 3
? -1
3
3
3
0
READY
```

The results are obviously not correct. The program printed out four numbers instead of the three we read in (−1 does not count as a valid number), and of those numbers which *should* have been valid, all appeared as equal to 3.

We can make a preliminary guess at the bug and say that it might either be an indexing problem, since all three values are identical; or the logic of the sorting routine itself is in error.

To successfully make use of a print statement, we must first answer these two questions: What variables would we like to see displayed? Where in the program is the best (most informative) place to grab these values?

Based upon our initial hunch as to what might be causing the bug in the sorting program, we would probably want to take a look at the index variables *J* and *K*, since they are the variables related to arrays B and A, respectively. And, since the values contained in arrays A and B are essentially at the heart of the sorting program, we will also want to examine them during program execution.

The best place to print out the index values is at the beginning of the innermost loop:

```
120 FOR J=1 TO I
130 FOR K=J TO I
→ INSERT PRINT STATEMENT HERE ←
140 IF B(J)>=A(K) THEN GOTO 170
150 LET B(J)=A(K)
```

```
160 GOTO 130
170 NEXT K
180 NEXT J
```

The innermost loop encompasses only three lines of code, none of which appear to affect the value of either index variable *J* or *K*. Therefore, printing out the values of *J* and *K* right after line 130 will most likely yield reliable results.

Where to print out the A and B array values and how many values from each array to print each time require careful consideration. We could, for instance, print out every element of both arrays each time the program made a transit through loop 3 (see Fig. 3-1). This is certainly the most complete approach, but it may also be the most inefficient: both arrays A and B contain 100 elements each. It would be very difficult, if not impossible, to fit all 100 elements of one of the arrays on to the CRT screen at one time, and it would definitely be impossible to do so for all elements of both arrays for more than one cycle.

Instead, we elect to display the current values of the index variables and the elements which these variables point to in arrays A and B. This reduces our output to only four values per loop and brings down the total volume of output to a manageable level, assuming we continue to test the program with a small sampling of input variables.

As for the placement of the array element print statement, we could try making it line 165, immediately before the ending statement of the innermost loop. By putting the print statement at the end of the loop we may be able to follow the progress of how values are loaded into array B from array A.

If we insert the following two statements:

```
135 PRINT "J,K"; J; K
165 PRINT "B(J), A(K)"; B(J); A(K)
```

and then run the program, we would get:

```
RUN
? 1
? 2
? 3
? -1
J,K 1 1
J,K 1 2
```

```
J,K 1 3
J,K 1 4
J,K 2 2
J,K 2 3
J,K 2 4
J,K 3 3
J,K 3 4
J,K 4 4
3
3
3
0
READY
```

Apparently one of our print statements did not get executed. A listing of the sorting section of the code reveals why:

```
120 FOR J=1 TO I
130 FOR K=J TO I
135 PRINT "J,K"; J; K
140 IF B(J)>=A(K) THEN GOTO 170
150 LET B(K)=A(K)
160 GOTO 130
165 PRINT "B(J),A(K)"; B(J); A(K)
170 NEXT K
180 NEXT J
```

The program never reaches line 165. It is directed to either skip over it:

```
140 IF B(J)>=A(K) THEN GOTO 170
```

or branch back to the beginning of the loop before ever reaching line 165:

```
160 GOTO 130
```

This illustrates one of the main considerations when using print

statements: Correct placement is very important, and making such a determination is not always an automatic procedure.

Referring to the flowchart in Fig. 3-1, it looks as though a print statement would be successful placed anywhere along the return line leading into the increment box of the loop symbol. Such is not the case, however, as we found out in the previous examples. This would not be the first time the structure of a coded program did not perform exactly as the corresponding flowchart would indicate. Rather than being a curious anomaly, this is a sign of sloppy programming and should be corrected.

For the moment, however, rather than modify the existing code before being entirely sure of what exactly it is doing, we decide to move the A(*K*) and B(*J*) print statement to another location.

We might move the array printout up to join with the index variable printout at line 135, but we should first convince ourselves that the printout is working correctly. Looking at our diagnostic print, we have:

```
J,K 1 1
J,K 1 2
J,K 1 3
J,K 1 4
J,K 2 2
J,K 2 3
J,K 2 4
J,K 3 3
J,K 3 4
J,K 4 4
```

When *J* equals 1, *K* ranges from 1 to 4; when *J* equals 2, *K* ranges from 2 to 4; for *J* equal to 3, *K* equals 3 and 4; and for *J* equal to 4, *K* equals 4. This seems to be in accord with the ranges of the two index variables as specified in the program:

```
120 FOR J=1 TO I
130 FOR K=J TO I
```

We might note in passing and for future reference that the index variable for the innermost loop, *K*, is itself indexed upon the index variable of the outermost loop, *J*. This may be intentional or it may be a bug. Hopefully, our print statement debugging method will tell us which it is.

Aside from our question about the specific choice of end points for

the *J* and *K* index variables, we may assume that the print statement which is supposed to display those values does in fact work. Therefore, moving the array printouts to the same point in the code would probably work correctly also.

Deleting line 165 and modifying line 135 results in:

```
135 PRINT "J,K,B(J),A(K)"; J;K;B(J);A(K)
```

Running the program with this modification yields:

```
RUN
YOU MAY ENTER UP TO 100 NOS, -1 ENDS LIST
? 1
? 2
? 3
? -1
J,K,B(J),A(K) 1 1 0 1
J,K,B(J),A(K) 1 2 1 2
J,K,B(J),A(K) 1 3 2 3
J,K,B(J),A(K) 1 4 3 -1
J,K,B(J),A(K) 2 2 0 2
J,K,B(J),A(K) 2 3 2 3
J,K,B(J),A(K) 2 4 3 -1
J,K,B(J),A(K) 3 3 0 3
J,K,B(J),A(K) 3 4 3 -1
J,K,B(J),A(K) 4 4 0 -1
3
3
3
0
READY
```

The index values are as before, and at last we are able to see what is contained in the A and B arrays.

If we now take the time to analyze the short table of printout that we have just generated, we will realize that one simple print statement has yielded much useful information. For example, we can follow each element of the B array as it is filled. Suppose we arrange the data a little differently. Consider the case when the outermost loop index is equal to 1, and we are cycling through the innermost loop as it increments from the

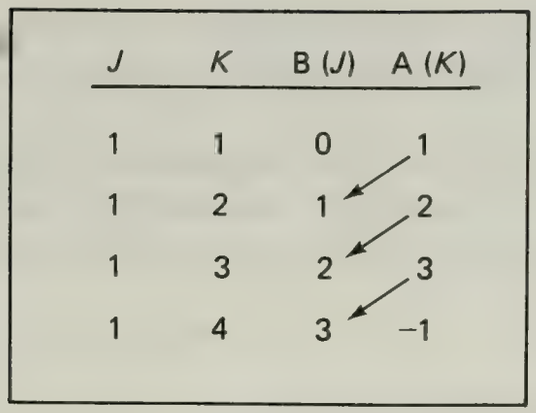


FIG. 3-2. PRINT statement data for sorting routine, first traverse through outermost loop.

starting value of *J* (in this case, 1) to the terminal value of *I* (always equal to 4), as in Fig. 3-2.

We can see that for the first loop the code is performing as it should. Initially, B(1) is equal to 0 and A(1) is equal to 1. The code falls through the inside loop; and when it makes its way back to the beginning of the loop, B(1) has been assigned the value of A(1)—which is as it should be, since 1 is greater than 0.

After every subsequent traverse through the inner loop, B(1) either retains its old value or takes on the value of the current element from array A (A(*I*)), depending upon which is the greater. Since at the end of the last time through the inner loop, 3 is greater than −1, B(1) finished the inner loop equal to 3.

Figure 3-3 shows what happens during the second cycle through the outer loop, when *J* is equal to 2. In this instance *K* ranges from 2 to 4, since it begins with the value of *J*, which is 2. Whereas in the first iteration over *J* the element from B was compared against all four elements of A, this time the new element from B is only being tested against three members from A.

The logic of the method makes sense: if during the first iteration the largest of *N* numbers contained in array A was discovered and placed into the first element of B, then during the second iteration we need only concern ourselves with *N*-1 numbers in array A. Likewise, in each subse-

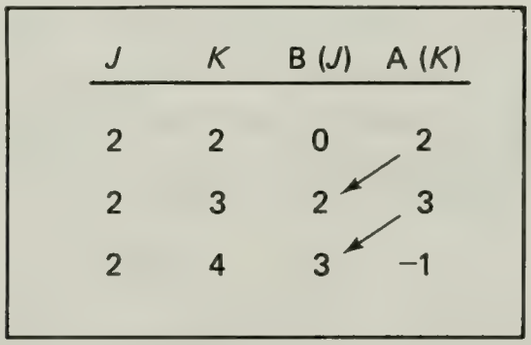


FIG. 3-3. PRINT statement data for sorting routine, second traverse through outermost loop.

quent iteration we can shorten the search by ignoring those numbers already sorted out and concentrating on the ever decreasing list of unsorted numbers.

Unfortunately, the actual coding does not coincide with the suggested logic. The way the code is structured, a correct answer will only result if the numbers are already in a sorted sequence, as follows:

```
RUN
YOU MAY ENTER UP TO 100 NOS, -1 ENDS LIST
? 3
? 2
? 1
? -1
J,K,B(J),A(K) 1 1 0 3
J,K,B(J),A(K) 1 2 3 2
J,K,B(J),A(K) 1 3 3 1
J,K,B(J),A(K) 1 4 3 -1
J,K,B(J),A(K) 2 2 0 2
J,K,B(J),A(K) 2 3 2 1
J,K,B(J),A(K) 2 4 2 -1
J,K,B(J),A(K) 3 3 0 1
J,K,B(J),A(K) 3 4 1 -1
J,K,B(J),A(K) 4 4 0 -1
3
2
1
0
READY
```

COMPARING INDEX VARIABLES

We must do two things if we are to debug the sorting program: (1) Since the numbers in array A are in an arbitrary order (which is the whole point of having a sorting routine), we must compare every value of A against the current value from B. The way to fix this is to have K range from 1 to *I*:

```
130 FOR K=1 TO I
```

(2) We require some method of eliminating from consideration those values from array A which have already been sorted out and transferred to array B. This could be as simple as making those values zero, but we first need a way of identifying which elements of A are to be zeroed out. Line 140 has some of the qualities we seek:

```
140 IF B(J)>=A(K) THEN GOTO 170
```

This line of code says that if the maximum has already been found, then we are to pass over the particular number under consideration and move on to the next. Note, however, that line 140 is testing for two different properties of the number from array A. Line 140 asks if that number is either greater than or equal to the current number from array B. We can use this line of code to our advantage by breaking it up into two separate statements:

```
140 IF B(J)>A(K) THEN GOTO 170
145 IF B(J)=A(K) THEN LET A(K)=0
```

Clearly, if B(*J*) is greater than A(*K*), A(*K*) is not a new maximum, and we must move on to the next value. However, if B(*J*) equals A(*K*) then we have found the maximum value. It should be zeroed out in array A so as not to cause bugs later on.

Also, we should delete line 160:

```
160 GOTO 130
```

This line, in effect, is circumventing the termination statement of a FOR-NEXT loop, and while no serious bugs seem to have resulted, it is very poor programming practice and should be avoided.

A complete listing of the program as currently modified is as follows:

```
5 REM PROGRAM TO SORT NUMBERS
10 REM SET UP ARRAYS
20 DIM A(100), B(100)
30 PRINT "YOU MAY ENTER UP TO 100 NOS, -1 ENDS LIST
40 REM INITIALIZE INDEX
50 LET I=1
60 REM READ DATA, STOP IF = -1
```

```
70 INPUT A(I)
80 IF A(I)=1 THEN GOTO 110
90 LET I=I+1
100 GOTO 70
110 REM SORT LIST OF NUMBERS
120 FOR J=1 TO I
130 FOR K=1 TO I
140 IF B(J)>A(K) THEN GOTO 170
145 IF B(J)=A(K) THEN LET A(K)=0
150 LET B(J)=A(K)
170 NEXT K
180 NEXT J
190 REM PRINT OUT ANSWER
200 FOR L=1 TO I
210 PRINT B(L)
220 NEXT L
230 END
```

Running the program this time results in:

```
RUN
YOU MAY ENTER UP TO 100 NOS, −1 ENDS LIST
? 1
? 2
? 3
? −1
J,K,B(J),A(K) 1 1 0 1
J,K,B(J),A(K) 1 2 1 2
J,K,B(J),A(K) 1 3 2 3
J,K,B(J),A(K) 1 4 3 −1
J,K,B(J),A(K) 2 1 0 1
J,K,B(J),A(K) 2 2 1 2
J,K,B(J),A(K) 2 3 2 3
J,K,B(J),A(K) 2 4 3 −1
J,K,B(J),A(K) 3 1 0 1
J,K,B(J),A(K) 3 2 1 2
J,K,B(J),A(K) 3 3 2 3
```

```
J,K,B(J),A(K) 3 4 3 −1
J,K,B(J),A(K) 4 1 0 1
J,K,B(J),A(K) 4 2 1 2
J,K,B(J),A(K) 4 3 2 3
J,K,B(J),A(K) 4 4 3 −1
3
3
3
3
READY
```

By changing the limits of index variable *K* to range from 1 to *I* instead of from *J* to I, we have caused a 60% increase in the length of the diagnostic printout.

But by splitting the "greater than or equal to" test of line 140 into two separate tests, we do not seem to have accomplished anything at all. Why? The diagnostic printout, as reproduced in Fig. 3-4, tells the story.

The innermost loop is now comparing each element of array A against the current value from array B and the greatest value from A is being fed to B, just as it should be. The bug which we see manifested in the "answer" printout and which is made glaringly visible in the diagnostic print statement, is that array A remains unaffected through the entire sorting process. We should see the contents of array A being replaced with zeros as the numbers are pulled out in decreasing order.

Referring to the listing, we see why:

```
120 FOR J=1 TO I
130 FOR K=1 TO I
135 PRINT "J,K,B(J),A(K)"; J; K; B(J); A(K)
140 IF B(J)>A(K) THEN GOTO 170
145 IF B(J)=A(K) THEN LET A(K)=0
150 LET B(J)=A(K)
170 NEXT K
180 NEXT J
```

Line 145 will zero out the element in array A if it is equal to the current maximum as represented by the element from array B. However, as we can see from the diagnostic printout, at any particular combination of *J* and *K* values, B(*J*) and A(*K*) are never equal.

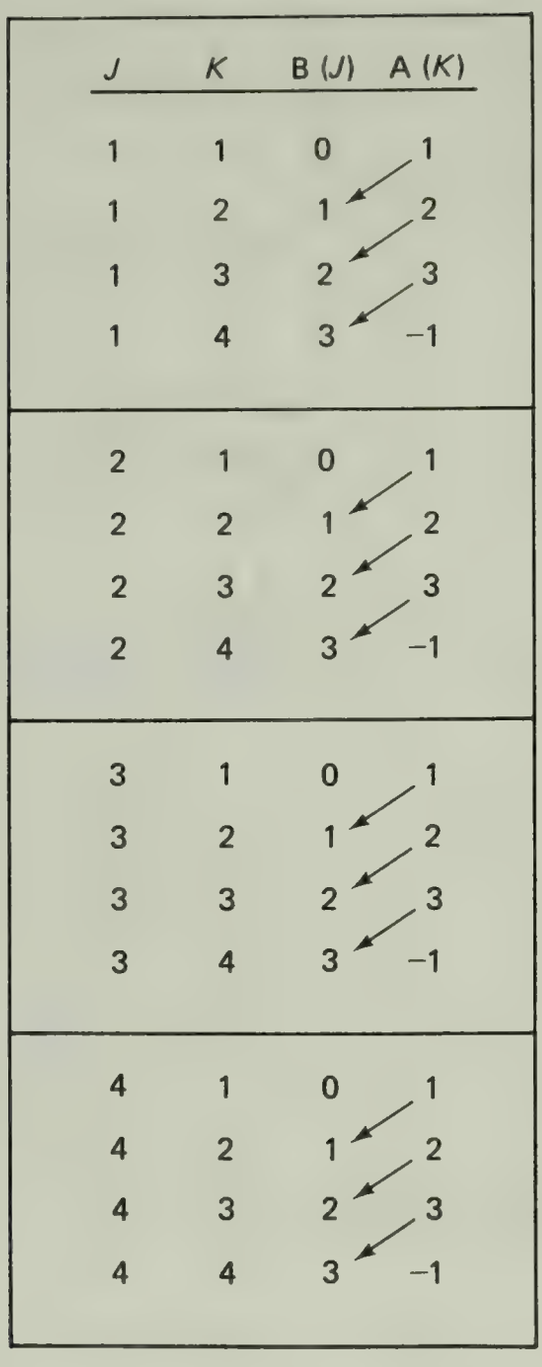


FIG. 3-4. Diagnostic printout for sorting program, after first round of modifications.

What if we were to move line 145 down to line 155? Certainly B(*J*) would then equal A(K), because they were just made equal at line 150. The error in this reasoning should be obvious, but it is easily demonstrated by making the change and running the program:

```
155 IF B(J)=A(K) THEN LET A(K)=0
DELETE 145
RUN
YOU MAY ENTER UP TO 100 NOS, −1 ENDS LIST
? 1
? 2
```

```
? 3
? -1
J,K,B(J),A(K) 1 1 0 1
J,K,B(J),A(K) 1 2 1 2
J,K,B(J),A(K) 1 3 2 3
J,K,B(J),A(K) 1 4 3 -1
J,K,B(J),A(K) 2 1 0 0
J,K,B(J),A(K) 2 2 0 0
J,K,B(J),A(K) 2 3 0 0
J,K,B(J),A(K) 2 4 0 -1
J,K,B(J),A(K) 3 1 0 0
J,K,B(J),A(K) 3 2 0 0
J,K,B(J),A(K) 3 3 0 0
J,K,B(J),A(K) 3 4 0 -1
J,K,B(J),A(K) 4 1 0 0
J,K,B(J),A(K) 4 2 0 0
J,K,B(J),A(K) 4 3 0 0
J,K,B(J),A(K) 4 4 0 -1
3
0
0
0
READY
```

For *J* equal to 1 (the first time through the outermost loop), *K* ranged from 1 to 4. The first transit through the inside loop, *J*=1 and *K*=1; also, B(1)=0 and A(1)=1. Since 1 is greater than 0, the test at line 140 failed:

```
140 IF B(J)>A(K) THEN GOTO 170
```

A(1) was therefore assumed to be the new maximum, and B(1) was set equal to it at line 150.

```
150 LET B(J)=A(K)
```

Immediately thereafter, line 155, the line which we shuffled around, promptly replaced A(1) with a 0.

```
155 IF B(J)=A(K) THEN LET A(K)=0
```

This process repeated itself for all the numbers in array A, one after the other, except for the last value, −1. The value −1, being less than all the other numbers, including 0, would be the only value to pass the test at line 140.

We note that if the list of numbers read into the program had *not* been in strictly ascending order, a significantly different outcome would have resulted, as for example (minus the diagnostic printout):

```
RUN
? 4
? 10
? 5
? 1
? 6
? −1
10
6
1
0
0
0
READY
```

Judging by the output just generated we might come to a completely different (and incorrect) conclusion about the nature of the bug. At the very least, we would waste a greater amount of time in general head-scratching. When debugging a program, the guiding principle should be to simplify as much as possible; try to get the maximum amount of *useful* information out of every technique employed.

We are now nearing the point of final resolution for this particular problem. As we pointed out earlier, we need a way of unambiguously establishing which element of array A, for any given iteration over the innermost loop, is the current maximum. Once identified, we will also want to remove that element from further consideration.

We were correct in believing that the test at line 140 was a strong clue towards resolving our dilemma:

```
140 IF B(J)>=A(K) THEN GOTO 170
150 LET B(J)=A(K)
```

If B(*J*) already contains the maximum value to be found during the

current iteration, then whatever quantity is represented by A(K) will be less than B(J). The test will be true and the program will move on to the next value in array A. However, if B(J) is less than the value of A(K), then B(J) will be updated to be equal to A(K), the new current maximum.

We cannot zero out any A array values within the innermost loop since, as we learned previously, unpredictable results will occur unless the data already exists in order of decreasing value. The only safe place to perform such "zeroing out" is outside the bounds of the innermost loop but before the next transit of the outermost loop, as in Fig. 3-5.

In terms of coding, this would be expressed as:

```
175 LET A(T)=O
```

IDENTIFYING INDEX VARIABLES

The only thing left to resolve is the identity of the index variable T. T must point to that element of array A which has been identified as the current maximum value sorted out during the last cycle of iteration

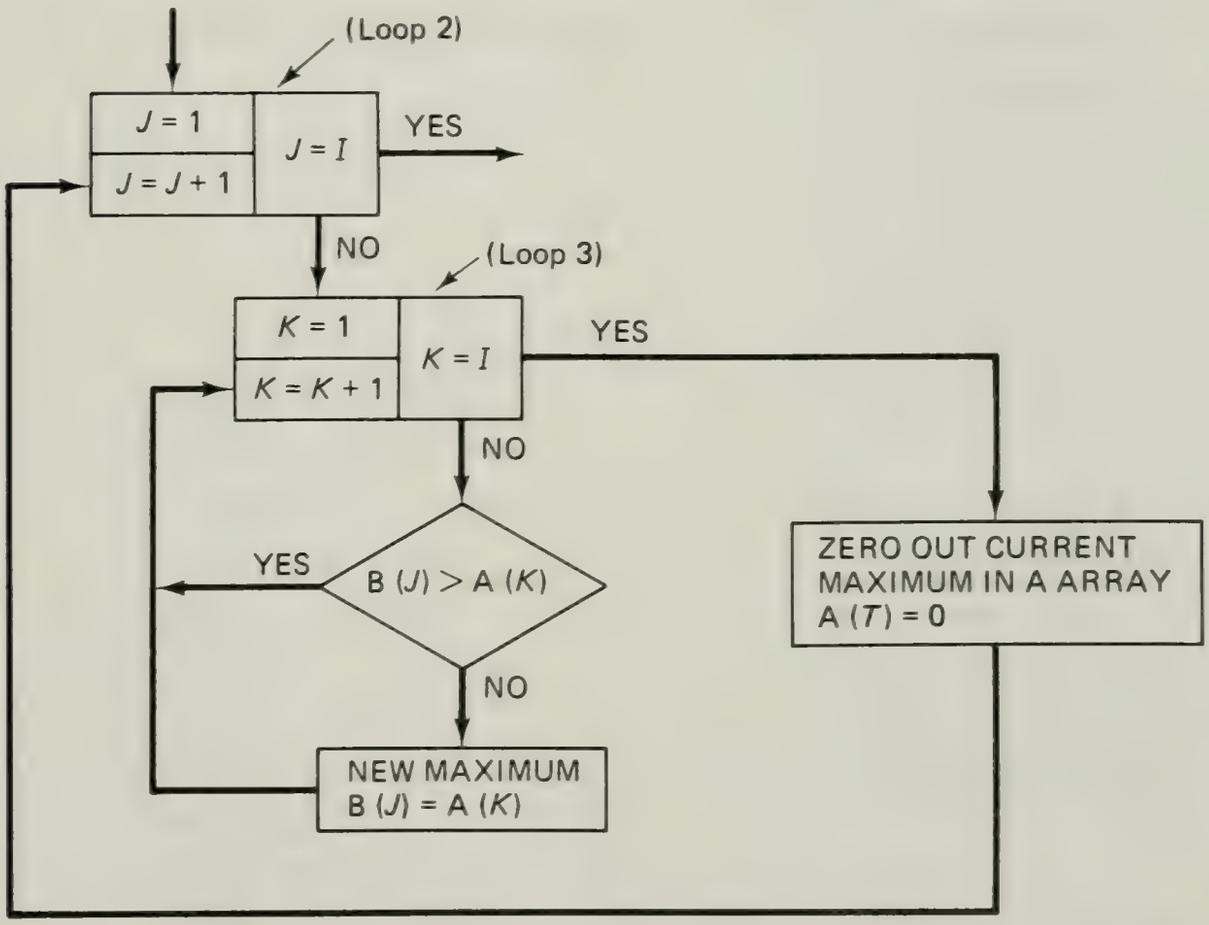


FIG. 3-5. Portion of number-sorting program flowchart, showing correct location of coding to zero out element of array A identified as current maximum value. Loop 2—outermost loop; loop 3—innermost loop.

through the innermost loop. The index variable *K* also points to members of array A, and during one cycle through the innermost loop it also points to the current maximum value in A.

We can preserve the identity of *K* by equating it to a storage variable such as *T*. The proper place for this would be following line 150, the place where the current maximum value itself is saved:

```
150 LET B(J)=A(K)
155 LET T=K
```

Running the program with these final modifications would yield:

```
RUN
YOU MAY ENTER UP TO 100 NOS, -1 ENDS LIST
? 1
? 2
? 3
? -1
J,K,B(J),A(K) 1 1 0 1
J,K,B(J),A(K) 1 2 1 2
J,K,B(J),A(K) 1 3 2 3
J,K,B(J),A(K) 1 4 3 -1
J,K,B(J),A(K) 2 1 0 1
J,K,B(J),A(K) 2 2 1 2
J,K,B(J),A(K) 2 3 2 0
J,K,B(J),A(K) 2 4 2 -1
J,K,B(J),A(K) 3 1 0 1
J,K,B(J),A(K) 3 2 1 0
J,K,B(J),A(K) 3 3 1 0
J,K,B(J),A(K) 3 4 1 -1
J,K,B(J),A(K) 4 1 0 0
J,K,B(J),A(K) 4 2 0 0
J,K,B(J),A(K) 4 3 0 0
J,K,B(J),A(K) 4 4 0 -1
3
2
1
0
READY
```

It looks as though the sorting program finally works, but we still have a cosmetic bug hanging on. Because the program must read in a −1 value as a signal to begin processing, it thinks that it has four values to sort instead of the actual three. Why? Because the index value *I* must be updated *before* a read statement is executed; otherwise the new value would write on top of a previously read-in value. The cure for this bug is simple and requires only that we reduce the value of *I* by one after the terminating −1 is read in:

```
80 IF A(I)=−1 THEN GOTO 110
—
—
110 REM SORT LIST OF NUMBERS
115 LET I=I−1
```

Can we say that the program is now debugged? We can say that for the values 1, 2, and 3, in that order, the program appears to be debugged. A more general test would be useful, however:

```
RUN
YOU MAY ENTER UP TO 100 NOS, −1 ENDS LIST
? 45
? 2.7
? 142
? 6
? 17
? −1
J,K,B(J),A(K) 1 1 0    45
J,K,B(J),A(K) 1 2 45   2.7
J,K,B(J),A(K) 1 3 45   142
J,K,B(J),A(K) 1 4 142 6
J,K,B(J),A(K) 1 5 142 17
J,K,B(J),A(K) 2 1 0    45
J,K,B(J),A(K) 2 2 45   2.7
J,K,B(J),A(K) 2 3 45   0
J,K,B(J),A(K) 2 4 45   6
J,K,B(J),A(K) 2 5 45   17
J,K,B(J),A(K) 3 1 0    0
J,K,B(J),A(K) 3 2 0    2.7
J,K,B(J),A(K) 3 3 2.7  0
```

```
J,K,B(J),A(K) 3 4 2.7  6
J,K,B(J),A(K) 3 5 6    17
J,K,B(J),A(K) 4 1 0    0
J,K,B(J),A(K) 4 2 0    2.7
J,K,B(J),A(K) 4 3 2.7  0
J,K,B(J),A(K) 4 4 2.7  6
J,K,B(J),A(K) 4 5 6    0
J,K,B(J),A(K) 5 1 0    0
J,K,B(J),A(K) 5 2 0    2.7
J,K,B(J),A(K) 5 3 2.7  0
J,K,B(J),A(K) 5 4 2.7  0
J,K,B(J),A(K) 5 5 2.7  0
142
45
17
6
2.7
READY
```

The answer is correct, but the volume of diagnostic printout has increased significantly. If we were to attempt to debug the program using a sample of 10 numbers, we would have to wade through 100 lines of print: the innermost loop executes 10 times for every single traverse of the outer loop, and the outer loop itself cycles through 10 times (10 times 10 equals 100).

This points up something that was mentioned before: *placement is critical.* Many times a bug can be traced to the vicinity of a series of concentric loops, and in that case the programmer faces a real challenge in determining when and where to make use of the print statement.

For example, consider the following excerpt from a geographical mapping program:

```
1010 REM PROCESS 100 X 100 MATRIX
1020 REM OF ALTITUDE VALUES
1030 REM IF VALUE IS MISSING
1040 REM INTERPOLATE FROM NEIGHBORS
1050 FOR I=1 TO 100
1060 FOR J=1 TO 100
1070 ON L5 GOTO 2030, 1090, 1235
1080 IF Z*R(J)>SQR(M/L9) THEN GOTO 1410
```

```
1090 REM INTERPOLATE VALUES
1100 FOR K=1 TO 8
1110 LET T1=M(I,J)*M(I-1,J)*M(I+1,J)/3
1120 LET T2=M(I-1,J+1)*M(I,J)*M(I+1,J-1)/3
1130 LET T3=M(I-1,J-1)*M(I,J)*M(I+1,J+1)/3
1140 LET T4=M(I,J+1)*M(I,J)*M(I,J-1)/3
1150 LET N=T1*T2*T3*T4/4
1160 NEXT K
```

There is enough going on in this one small section of code to generate literally thousands of lines of output from one print statement. In general it is not wise to place a print statement buried deep within a series of nested loops. If a print statement *must* be located inside a number of nested loops, it is advisable to put some kind of a clamp or switch on the statement so that it is only activated when there is a high probability that it will yield useful information. As an illustration, if we wanted to use a print statement to debug the following example, instead of writing

```
1155 PRINT"I,J,K,T1,T2,T3,T4,N"; I;J;K;T1;T2;T3;T4;N
```

we would be better off to write

```
1155 IF N<=0 THEN PRINT "I,J,K,T1,T2,T3,T4,N"; I;J;K;T1;T2;T3;T4;N
```

The second method would cause the diagnostic to be printed only in those cases where *N* was less than or equal to zero, a sure sign of trouble. Had we not clamped the print statement, the sheer volume of output produced would have been like a smokescreen, making the debugging task that much more difficult.

Correctly used, the print statement is perhaps the easiest debugging tool available. It is quick to apply and can yield much (sometimes too much) potentially useful information. The print statement is well suited to an interactive approach to debugging, which of course implies access to a functioning computer as a condition of its use.

4

Adding Program Patches

There are situations when, despite our efforts to design and execute what we consider to be an intelligent and well-structured program, bugs appear that are either very elusive to understand or are very difficult to fix. On such occasions the programmer must often resort to a technique known as *patching*. Usually, when we speak of debugging we refer to the process of identifying a bug, tracing it to its source, and then correcting the problem. When we resort to patching a program, however, we are treating the symptoms rather than the cause; patches are not therefore a debugging technique in the strictest sense, yet they find common use in many debugging efforts.

Patches may at first seem like the easy way out; but in fact, they require a firm understanding of the code they are being applied to. Suppose we traced a bug to the following fragment of code:

```
1720 LET X2=SQR(32*Y)/M
1730 IF X2>10 THEN GOSUB 1500
```

Suppose further that we discovered that the bug that caused the program to fail is that the variable *M* occasionally takes on a negative or zero

value. We might select any one of the following patches to use for the program:

1. 1715 IF M<=0 THEN LET M=1
2. 1715 IF M<=0 THEN LET M=−M
3. 1715 IF M<=0 THEN LET X2=1:GOTO 1730

The list of possible patches is almost limitless. We might have the program do any number of tasks if *M* is less than or equal to 0; but any change we make with our patch, even if it seems as harmless as setting *M* equal to 1, will have repercussions later on in the program. The challenge when using patches is to make a change that will not only cure the symptoms being caused by the bug, but that will have the fewest and least damaging effects at subsequent points in the program.

EXAMPLE: MONITORING ROOM TEMPERATURE

Consider the following situation:

> We have been asked by a local company to computerize the environmental control in its new office building. The building is divided into four sections, each with its own heating and air-conditioning units. The optimum environmental temperature for office work is 74°F.

Suppose we accept the assignment and, upon further investigation, learn that each of the four work areas has its own temperature sensor, which reads in degrees Fahrenheit, to give us our monitor input. We also learn that interfacing has already been taken care of; we need only to place a value on one of the SOL's output ports (as we did in Chapter 1 when we controlled the model railroad) and the rest will be taken care of.

With the preliminaries decided, our next step is to draw up a flowchart, as in Fig. 4-1. We note that the flowchart is very straightforward; the entire program is essentially no more than one great loop, endlessly cycling through itself. Each sensor in turn is queried as to the temperature of its control space, and that reading is compared with the status of the corresponding air-conditioning and heating units. Based on the sensor inputs and the heater and air-conditioner status flags, either the heater is turned on and the air conditioner is turned off, or the air conditioner is turned on and the heater is turned off, or no action at all is taken.

The program is as follows:

```
10 REM TEMPERATURE CONTROL PROGRAM
```

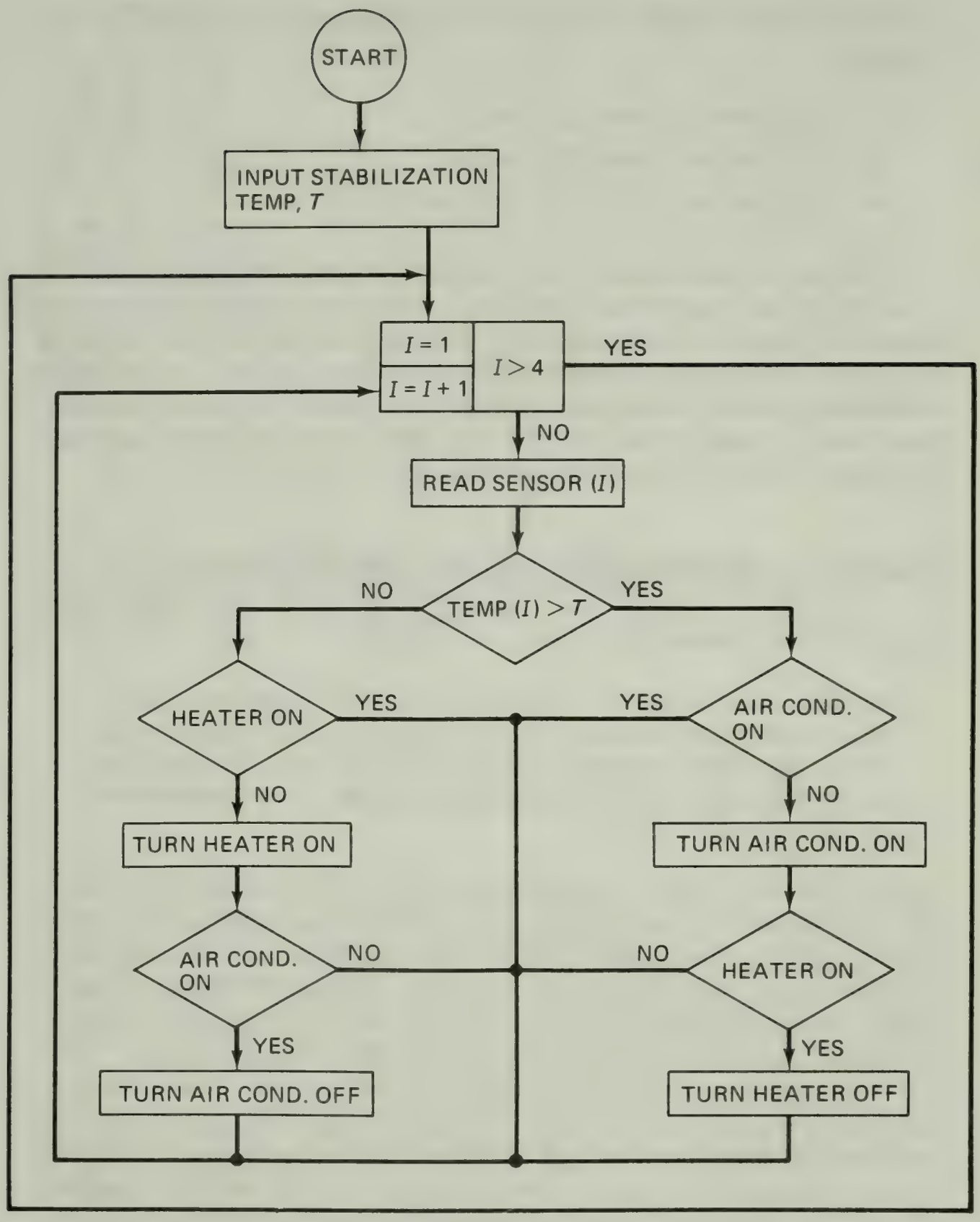


FIG. 4-1. Flowchart for environmental control program.

```
20 REM FOR ACME WIDGETS INC.
30 REM I=SENSOR INDEX
40 REM J=HEATER INDEX
50 REM K=COOLER INDEX
60 REM PORTS 1-4: SENSORS
```

```
70 REM PORTS 5-8:HEATERS
80 REM PORTS 9-12:COOLERS
90 PRINT "WHAT IS THE STABILIZATION TEMPERATURE?"
100 INPUT T
110 FOR I=1 TO 4
120 LET J=I+4
130 LET K=I+8
140 LET H=INP(J)
150 LET C=INP(K)
160 LET T2=INP(I)
170 IF T2>=T THEN GOTO 230
180 IF H=1 THEN GOTO 270
190 OUT J,1
200 IF C=0 THEN GOTO 270
210 OUT K,0
220 GOTO 270
230 IF C=1 THEN GOTO 270
240 OUT K,1
250 IF H=0 THEN GOTO 270
260 OUT J,0
270 NEXT I
280 GOTO 110
290 END
```

Tracing the program through, playing computer, we do not find any apparent bugs, and therefore load the computer with our program and type RUN. The interior temperature of the four offices starts out at 83°F, but as soon as our program takes over control, the air conditioner switches on and begins to cool things down. The president of Acme Widgets is happy, we receive a check for our labors, and everything looks like another success story for microcomputers.

Two hours later we get a phone call from the president of Acme Widgets saying that the offices have been alternately blasted with cold air and baked with hot air. To top things off, two of the air-conditioning units have broken down from apparent thermal overload.

Arriving at Acme Widgets, we hear the full story: For the first hour everything was fine as the air conditioners brought the temperature down to about 74°F. Suddenly one of the offices dropped below 74°, causing the heater to kick in to raise the temperature up to the stabilization point. The influx of warm air into the office displaced a large volume of cold air into

one of the neighboring offices, causing it to become cooler than 74°F. This in turn caused the second office heater to engage.

Meanwhile, back in the first office the temperature sensor had been so placed that by the time the senor registered the stabilization temperature of 74°, the office itself was considerably warmer than that figure. The end result: a rolling hot and cold cycle moving from office to office. As each office approached stabilization, the effects of the neighboring office would be felt by the sensors and the system would attempt to compensate. But owing to the placement of the sensors, the compensation became overcompensation, which only fed and perpetuated the oscillations. Eventually, the temperature shifts became so sudden that the computer program was switching the air-conditioning units in and out of service so quickly that the compressors overloaded and caused the circuit breakers to open.

Sheepishly, we return to our program.

One thing to remember when using a computer to control real-world events is that the computer moves very fast in comparison to most natural events. This is especially the case with environmental control applications: the computer is querying the temperature sensors many hundreds of times each minute to see if the temperature has changed, yet it may take minutes for the air in a room to reach some sort of temperature equilibrium. During this time, drafts or eddies may cause the local air temperature around the sensor to fluctuate either over or under the stabilization point, causing the program to switch in hot or cool air, depending upon the conditions.

The bug in the program, then, is that we have a set of input parameters which cannot be controlled. When dealing with a human operator we can always have a bit of code which (1) tests to see if the input is acceptable, and (2) notifies the operator to try again if it is not. The difficulty with an environmental control program is that the inputs received from the sensors are all valid, but the rapidity of their fluctuations from one state to another causes the system as a whole to be unstable.

A patch is a convenient way to handle such a problem. We know why the program is failing, but there is no way for us to correct the basic cause of the bug: the physical realities of air mass mixing and convection. We therefore attack the problem from another angle; we patch the program in such a way as to minimize the random nature of the input data.

For example, one patch would have as its aim to slow down the program's cycling speed so as to smooth over and ignore short-lived temperature fluctuations. We cannot actually slow down the speed of execution, but we have previously come across a technique designed to throw stumbling blocks in the execution path and slow things down that way:

```
115 PAUSE 10
```

The statement is telling the program to pause during execution for 10 tenths of a second (or 1 second). One way we might make use of such a statement would be to use the statement as numbered, perhaps with a longer delay time:

```
115 PAUSE 50
```

Now, whenever the program reaches the top of the loop it will pause for 5 seconds before taking any readings and making any determinations based upon those readings.

We recall that one of the major items of caution involved with using patches is that we must always be on the alert for the consequences of our actions. Putting a pause statement in the program will certainly slow the rate of program execution, but will it solve the problem? Unfortunately, when programming for a real-time control application we cannot always make a change to the program and then run it to try it out. In the present situation, for example, it would look very bad for us if we were to reappear at Acme Widgets with a proposed solution for the bug if the air-conditioning units overloaded a second time. We might not get another chance.

To correctly and safely use a patch, therefore, we must have a solid enough understanding of the code and its function to be able to predict with reasonable certainty and effects of any patch we might make.

Suppose we try to predict the results of a pause statement. When we first begin the office temperature program, it will ask for the stabilization temperature and then wait 5 seconds before making the first set of measurements. Once the measurements are in (in less than 0.1 second), the program will decide whether the heater or the air conditioner should be turned on—or, if already on, whether they should remain on or be turned off. Immediately, the program branches back to the beginning of the loop to begin another traversal.

Before beginning the next pass through the loop, the program will again pause for 5 seconds. At the end of that time it will make the next pass through the loop; this process will be repeated indefinitely until the computer is turned off.

The problem of short-lived temperature fluctuations is still with us, however. We have slowed down the pace of the program only in the sense that we have separated periods of action by periods of inaction. We have no evidence to suggest that the temperature fluctuations will occur only during the pauses we've placed in the program. Fluctuations might just as well occur during the regular-speed passes through the measurement-and-control loop. That being the case, we are back to where we started from.

What we need to debug this program is a way to smooth out the

minor temperature fluctuations to arrive at some average temperature reading for each sensor. One way to accomplish this might be:

```
153 LET T2=0
155 FOR L=1 TO 10
160 LET T2=INP(I)+T2
162 NEXT L
164 LET T2=T2/10
```

We now have the temperature reading set equal to the average of 10 consecutive readings. Still, however, the computer executes instructions so quickly that the 10 readings, which would require well under 1 second total to accomplish, might well be subject to the same short-lived temperature fluctuations. In order to spread our average out over a wider base, we could either take a much bigger sample—say 100 data points—or we could spread the 10 samples out over a longer time by using another pause, such as:

```
153 LET T2=0
155 FOR L=1 TO 10
157 PAUSE 10
160 LET T2=INP(I)+T2
162 NEXT L
164 LET T2=T2/10
```

By now we realize that physical phenomena, such as the air temperature in a group of offices, cannot be controlled as precisely as an event which takes place completely within the memory of the computer, such as the program in Chapter 2 which performed integer multiplication. We find, for example, that it is impractical to monitor room temperature on an instant-by-instant basis and to then apply corrections just as instantaneously, because the air mass in an office does not behave as a homogeneous whole.

Recognizing the limitation of our program, we also recognize that we would be unable to control the random natural events which in turn control the behavior of our program. The solution we adopt, then, is to use patches to treat the symptoms without correcting the causes of the bug.

ANOTHER EXAMPLE: DETERMINING STUDENTS' GRADES

Patches are so named because they are emergency stopgaps designed to bolster up "leaky" code. Frequently a program will be written that will function without problem for many applications—then suddenly a situation will arise which the designer had not anticipated, and the program will fail. Rather than rewrite the program (an alternative that would probably provide a stronger program but would also involve a fresh testing and debugging phase), the choice is made to code in a patch to take care of the special-case application and keep the program in service.

Consider the following example:

```
10 REM PROGRAM TO DETERMINE TEST GRADES
20 REM S=SCORE N=NUMBER AT THAT SCORE
30 DIM S(100), N(100)
40 LET I=1
50 INPUT "TEST SCORE?",S(I)
60 IF S(I)=900 THEN GOTO 100
70 INPUT "NUMBER RECEIVING THAT SCORE?",N(I)
80 LET I=I+1
90 GOTO 50
100 REM B1=SUM OF N*S
110 REM B2=SUM OF N
120 REM B3=SUM OF N*S*S
130 LET B1=0
140 LET B2=0
150 LET B3=0
160 FOR J=1 TO I
170 LET B1=B1+(S(J)*N(J))
180 LET B2=B2+N(J)
190 LET B3=B3+(S(J)*S(J)*N(J))
200 NEXT J
210 LET M=B1/B2
220 LET R=SQR((B3-B2*M*M)/(B2-1))
230 LET B=M+R
240 LET A=B+R
250 LET C=M-R
260 LET D=C-R
270 PRINT "A RANGES FROM";A;"TO 100"
```

```
280 PRINT "B RANGES FROM";B;"TO";A
290 PRINT "C RANGES FROM";C;"TO";B
300 PRINT "D RANGES FROM";D;"TO";C
310 PRINT "F RANGES FROM 0 TO";D
320 END
```

This program is supposed to determine the letter grade dividing lines for a sample of test scores (the flowchart for the program is in Fig. 4-2). The structure of the program is not complicated; pairs of grades and number of students receiving that grade are entered from the keyboard. The pairs of numbers continue to be accepted until a value of 900 is read in as a score. This terminates the data collection phase of the program and moves the processing to the data manipulation phase. In this phase a mean score *M* is calculated as well as the standard deviation from that score *R*. The letter grade dividing lines are calculated from the mean and the standard deviations.

Suppose we test the program on some typical test scores:

Data Set No. 1

Score	*Number of Students Receiving Score*
10	1
20	2
30	6
40	9
50	10
60	9
70	6
80	2
90	1

Running the program, we would find:

```
A RANGES FROM 84.5 TO 100
B RANGES FROM 67.3 TO 84.5
C RANGES FROM 32.7 TO 67.5
D RANGES FROM 15.5 TO 32.7
F RANGES FROM 0 TO 15.5
```

The numbers look reasonable, and if we transfer them to a graph as in Fig. 4-3 we can confirm their correctness. Figure 4-3 illustrates a very

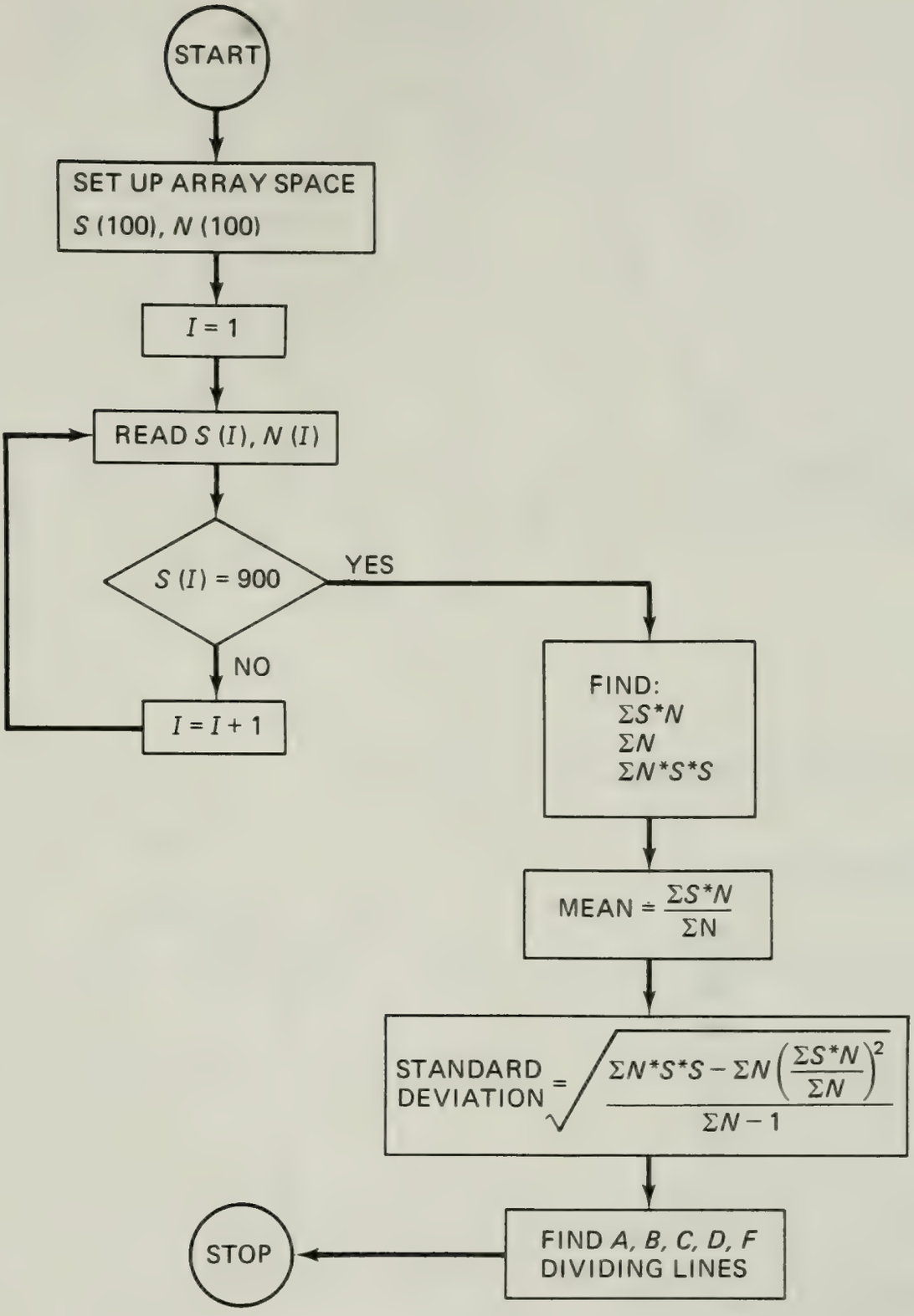
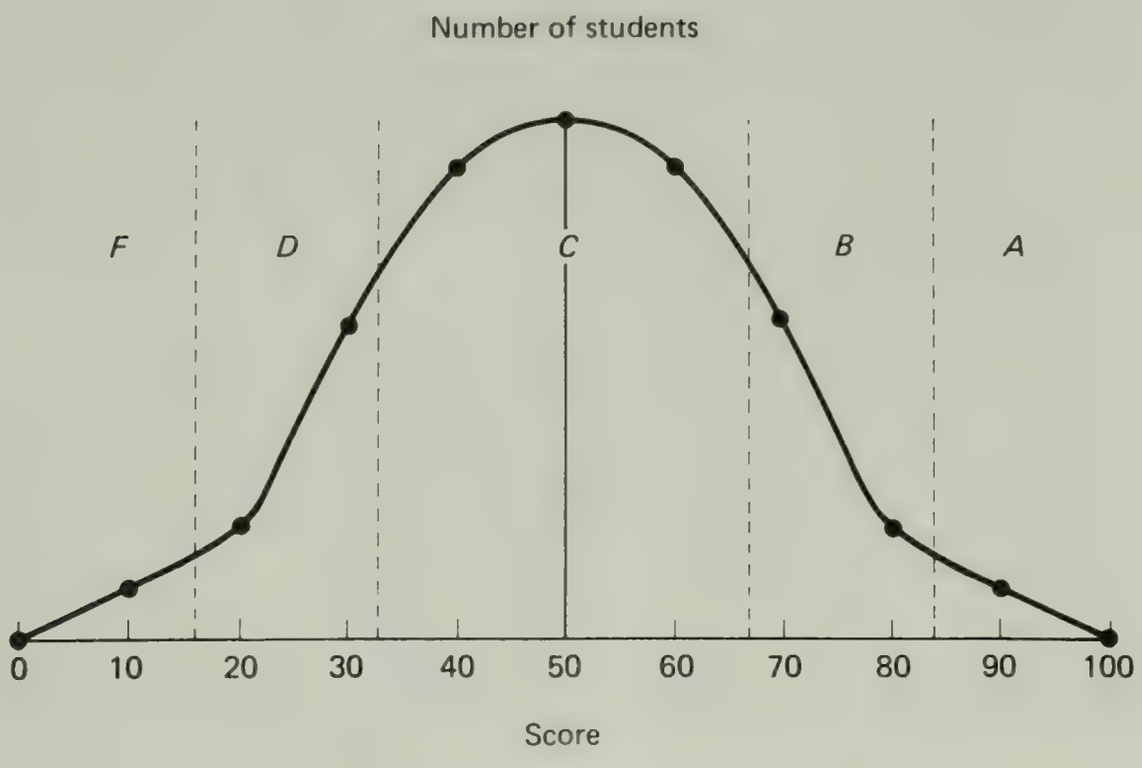


FIG. 4-2. Flowchart for letter grade determining program.

important point about this particular program: the grade-finding program is based on the assumption that every group of students' test scores will, when graphed, closely approximate the statistically average bell-shaped curve. The program works well because the assumption is almost always valid, even for most normally encountered deviations.



Consider the following two sets of data:

Data Set No. 2

Score	*Number of Students Receiving Score*
25	2
40	4
50	7
60	10
70	9
80	6
90	2

Data Set No. 3

Score	*Number of Students Receiving Score*
10	1
20	4
30	7
40	9
50	5
60	3
70	1
80	1

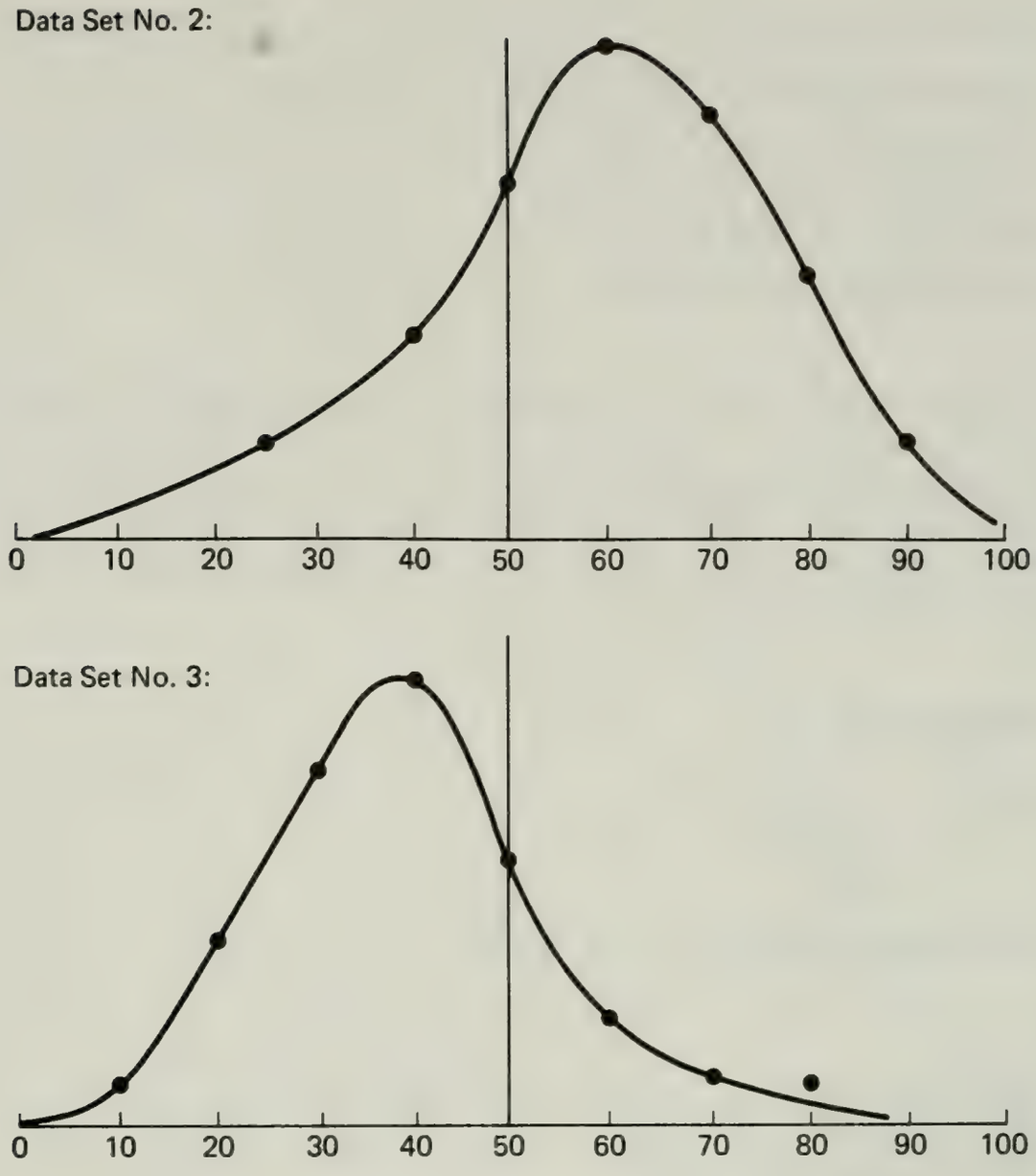


FIG. 4-4. Two examples of shifted data samples that are nonetheless handled successfully by the grading program.

Figure 4-4 illustrates what these two curves look like. We can see that while they both approximate the standard bell-shaped curve in general outline, neither one is centered around the score of 50, nor is either curve symmetrical. Nonetheless, we find upon running the program that it is sufficiently strong to deal with aberratrions such as these and still yield usable results:

For Data Set No. 1

A RANGES FROM 92.9 TO 100
B RANGES FROM 77.1 TO 92.9
C RANGES FROM 45.4 TO 77.1
D RANGES FROM 29.6 TO 45.4
F RANGES FROM 0 TO 29.6

For Data Set No. 3

A RANGES FROM 71.4 TO 100
B RANGES FROM 55.7 TO 71.4
C RANGES FROM 24.3 TO 55.4
D RANGES FROM 8.6 TO 24.3
F RANGES FROM 0 TO 8.6

Comparison of the three examples of data sets points up some interesting differences and similarities. We know from the program listing that the various letter-grade divisions are all computed with reference to *M*, the mean value, and *R*, the standard deviation. It is therefore an easy-enough exercise to work backwards from the printed output to determine what the values *M* and *R* were in each case.

Data Set No. 1

MEAN = (32.7+67.5)/2
= 100.2/2
= 50.1
STANDARD DEVIATION = 67.5−50.1
= 17.4

Data Set No. 2

MEAN = (45.4+77.1)/2
= 122.5/2
= 61.25
STANDARD DEVIATION = 77.1−61.25
= 15.75

Data Set No. 3

MEAN = (24.3+55.4)/2
= 79.7/2
= 39.85
STANDARD DEVIATION = 55.4−39.85
= 15.55

In all three cases the standard deviation is about 16 points in either direction from the mean value. The mean value, in contrast, is not at all the same value for each of the three examples, but its relevance to the data sets is clearly pointed out in Fig. 4-5, which superimposes the mean

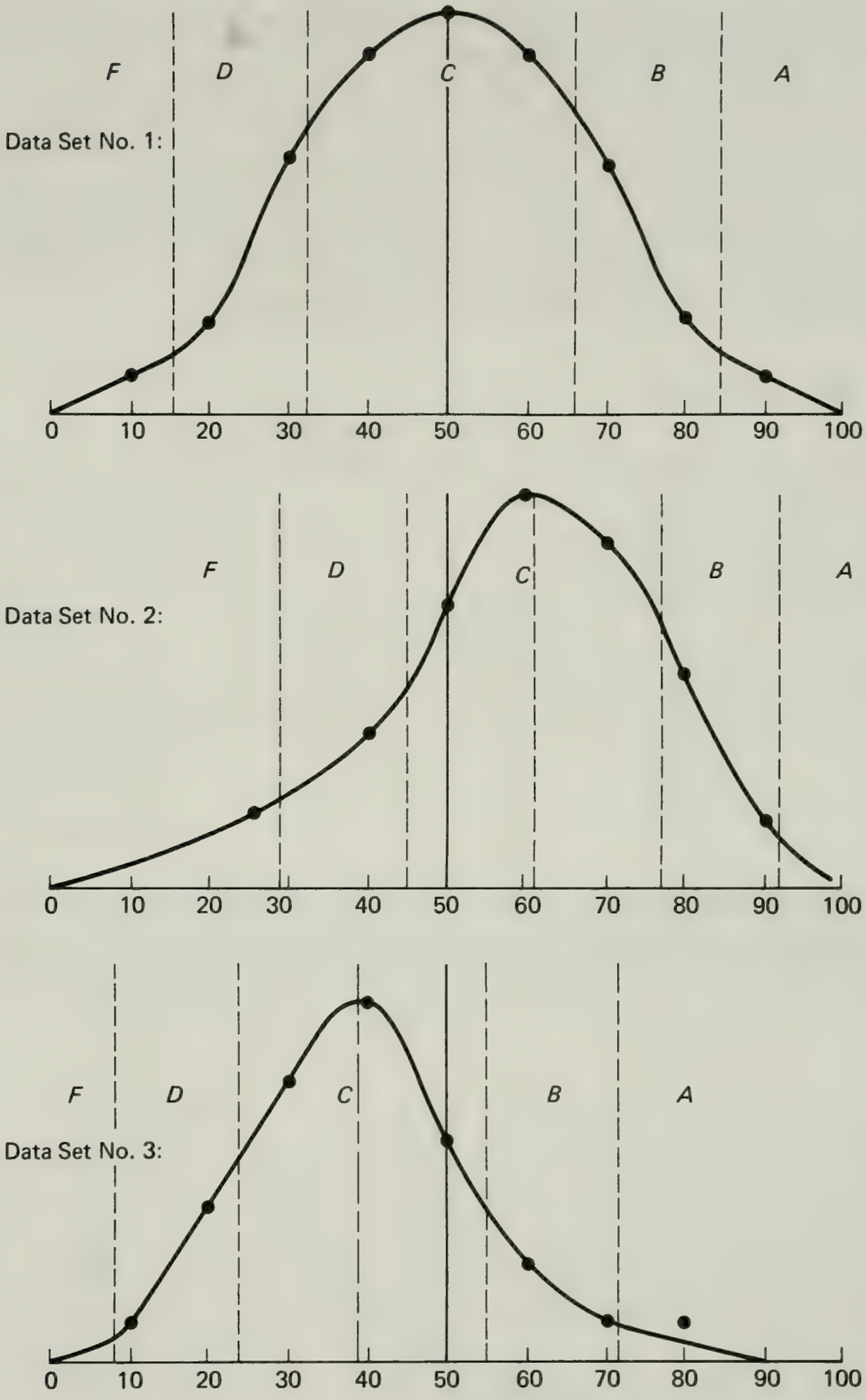


FIG. 4-5. Data Sets 1, 2 and 3 plotted with letter grade divisions superimposed. Note that in all cases the mean value closely approximates the maximum value.

value as well as the letter grade dividing lines on top of the already presented curves of the three data sets.

We note that the mean value seems to fall on or near the maximum point of each curve, as would seem intuitively correct. In actuality, the line drawn at the mean value divides the curve into the halves of equal area. Of what special significance is this to us? After all, the program makes the assumption (which we have found to be reasonable) that for most cases the data will assume the general outline of a symmetrical bell-shaped curve. But consider the following example:

Data Set No. 4

Score	*Number of Students Receiving Score*
20	1
30	2
40	2
50	3
60	4
70	6
80	9
90	8
95	5
100	1

The curve which represents this data is shown in Fig. 4-6. Note how the curve has been shoved over (skewed) onto the high end of the graph. In physical terms, the curve is telling us that for the examination in ques-

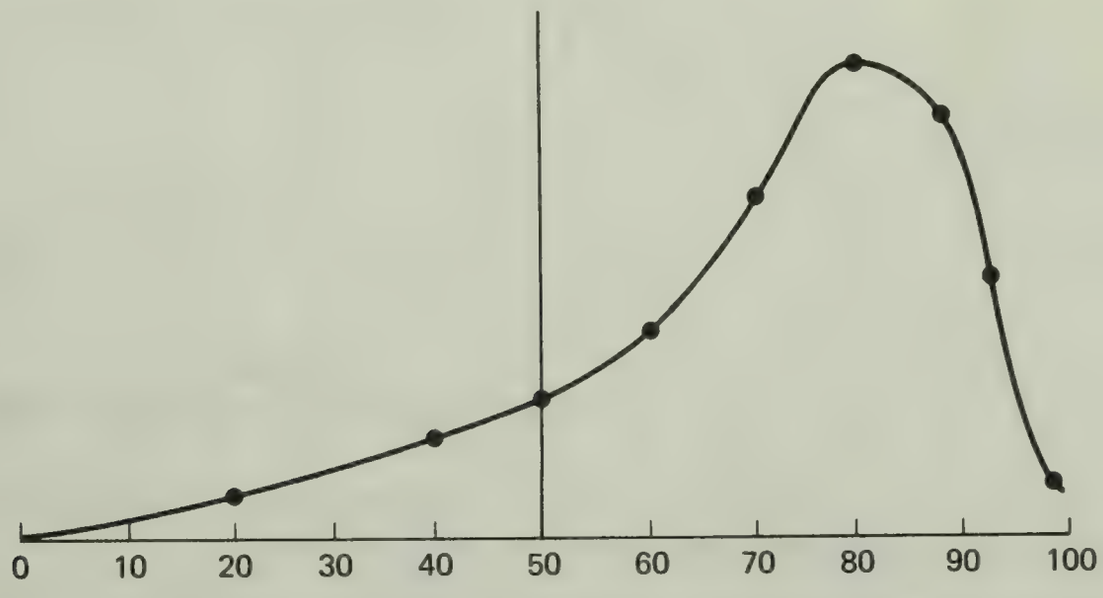


FIG. 4-6. The curve for Data Set No. 4. Note how the curve has been pushed over (skewed) to the high end of the graph.

tion, a disproportionate number of students scored very well. While the situation might be unusual it is by no means impossible and should therefore be provided for in the design of the grading program.

The output is as follows:

A RANGES FROM 113.9 TO 100
B RANGES FROM 93.4 TO 113.9
C RANGES FROM 52.2 TO 93.4
D RANGES FROM 31.7 TO 52.2
F RANGES FROM 0 TO 31.7

Obviously, this situation has not been provided for in the program. Mathematically, however, the program is correct; there is no bug of the sort we usually associate with malfunctioning programs. What we need is a patch to deal with the rare special occasion when data describing a highly skewed curve is fed into the program. The problem now is to determine what sort of patch should be instituted.

Suppose we begin by doing to this data set what we did to the three previous—working backwards from the printed result to discover the mean and standard deviation values:

Data Set No. 4

MEAN = (52.2+93.4)/2
= 145.6/2
= 72.8
STANDARD DEVIATION = 93.4−72.8
= 20.6

Immediately we can see that the standard deviation is not in line with the three previous examples, and a look at Fig. 4-7 tells even more. Before, the mean value calculated by the program had been very close to the maximum point on the curve. This was especially true for data set No. 1, which was a perfectly symmetrical group of data, and it continued to be true for data set Nos. 2 and 3 which, while not symmetrical, at least approximate a bell-shaped curve.

Data set No. 4, however, is very asymmetrical and the mean value reflects this deviation; it is significantly different from the maximum value of the curve. We now have a little more information than we did when we started, but the question still remains: What sort of patch should we use to correct the problem?

Actually, there is nothing we can do to modify the input to the pro-

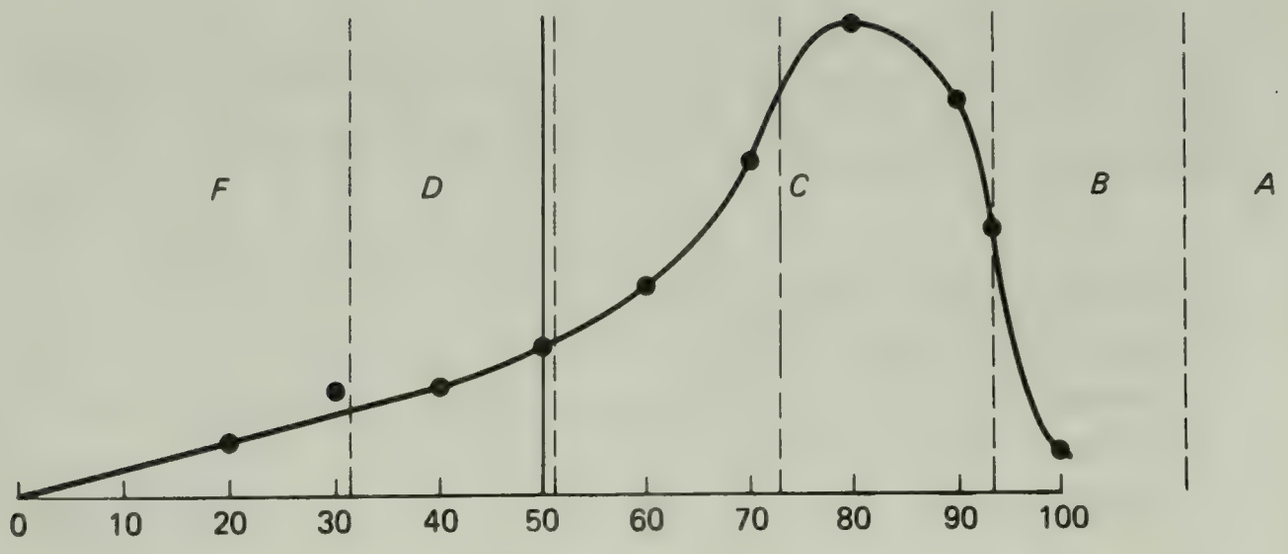


FIG. 4-7. Data Set No. 4 (skewed data) with grade divisions and mean value line superimposed. Note that the mean value is now significantly different from the maximum value (as compared to previous examples).

gram, since the input corresponds to a real event and cannot be modified. We might try some additional manipulation of the data once it is inside the program, but that would involve making additional assumptions (or modifying our original assumptions) regarding the statistical distribution of test scores and the validity of methods based upon those distributions to determine letter grades. The mean-versus-standard-deviation method is one well-accepted approach; we must consider how a potential patch would affect that method.

Suppose we were to attempt the patch diagrammed in Fig. 4-8. It keeps all of the values on-scale but note how artificial the top-end divi-

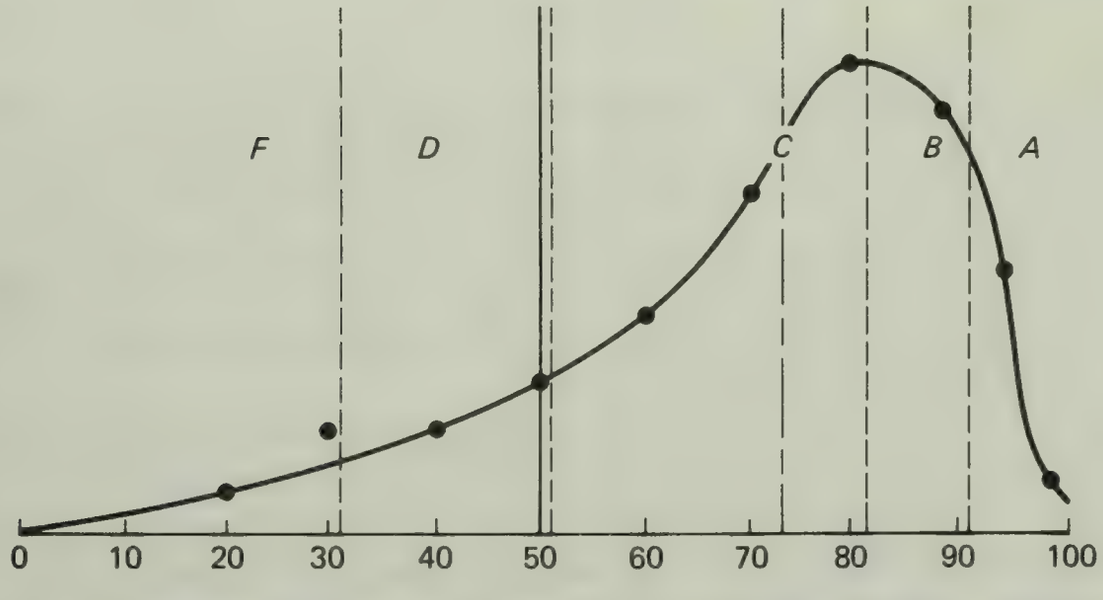


FIG. 4-8. Restructured grade divisions superimposed upon skewed data set. Note how the patch has artificially weighted the high end of the graph (above the mean score line).

sions look; we would have to determine if this is an acceptable deviation from the mathematically determined values. The danger here is that our patch must include a warning which could be printed out each time the special coding is invoked. If it does not, the figures thus computed, since they are apparently computed by a program based on a supposed fair and equitable method of grade determination, might assume an undeserved air of respectability—to someone's possible harm.

Patches, because they are usually written to address the special-case occurrence, should always be clearly identified both inside the program and next to any output which they produce. Thus:

```
261 IF A<=100 THEN GOTO 270
262 LET S=100-M
263 LET R2=S/3
264 LET B=M+R2
265 LET A=B+R2
266 PRINT "**NOTE**"
267 PRINT "DATA ORIGINALLY OUT-OF-RANGE"
268 PRINT "NEW BOUNDARIES COMPUTED"
269 PRINT "**NOTE**"
```

and the resulting printout for data set No. 4 would look like:

```
**NOTE**
DATA ORIGINALLY OUT-OF-RANGE
NEW BOUNDARIES COMPUTED
**NOTE**
A RANGES FROM 90.9 TO 100
B RANGES FROM 81.9 TO 90.9
C RANGES FROM 52.2 TO 81.9
D RANGES FROM 31.7 TO 52.2
F RANGES FROM 0 TO 31.7
```

Of course, not all patches must "fix" a particular program, especially when (as in this case) the patch makes such a severe modification of the program that the original rationale is sidetracked. In other words, it is sometimes most honorable if the patch does no more than admit defeat when defeat is obvious:

```
261 IF A<=100 THEN GOTO 270
```

```
262 IF D>=0 THEN GOTO 270
263 PRINT "SORRY, DATA IS TOO FAR OUT OF RANGE"
264 PRINT "AND CANNOT BE SUCCESSFULLY TREATED"
265 PRINT "SUGGEST YOU RESORT TO"
266 PRINT "HUMAN INTERPRETATION OF RESULTS"
267 GOTO 320
```

A FINAL EXAMPLE: TEACHING MUSICAL SCALES

Not all patches are written to help a program accommodate unexpected or uneven input from the real world, nor do all patches require potential compromise in the rationale of a particular program's construction. Consider the following program intended to teach musical scales. A look at the flowchart depicted in Fig. 4-9 helps to explain the program's operation.

```
10 DIM A(19),B(6)
20 DATA 67.32,67.35,68.98,68.32,68.35,69.98
30 DATA 69.32,69.35,70.32,70.35,71.98,71.32
40 DATA 71.35,65.98,65.32,65.35,66.98,66.32,66.35
50 DATA 3,3,2,3,3,3
60 FOR I=1 TO 19
70 READ A(I)
80 NEXT I
90 FOR J=1 TO 6
100 READ B(J)
110 NEXT J
120 PRINT "WELCOME TO LEARN-A-SCALE"
130 PRINT "REPRESENT SHARPS WITH A SHIFTED 3 (#)"
140 PRINT "AND FLATS WITH A LOWER CASE B (b)"
150 PRINT "WHAT SCALE WOULD YOU LIKE TO SEE?"
160 INPUT S$
170 LET S=LEN(S$)
180 IF S=2 THEN GOTO 210
190 LET T=ASC(S$)+.32
200 GOTO 260
210 LET T1$=S$(1)
220 LET T2$=S$(2)
230 LET T1=ASC(T1$)
```

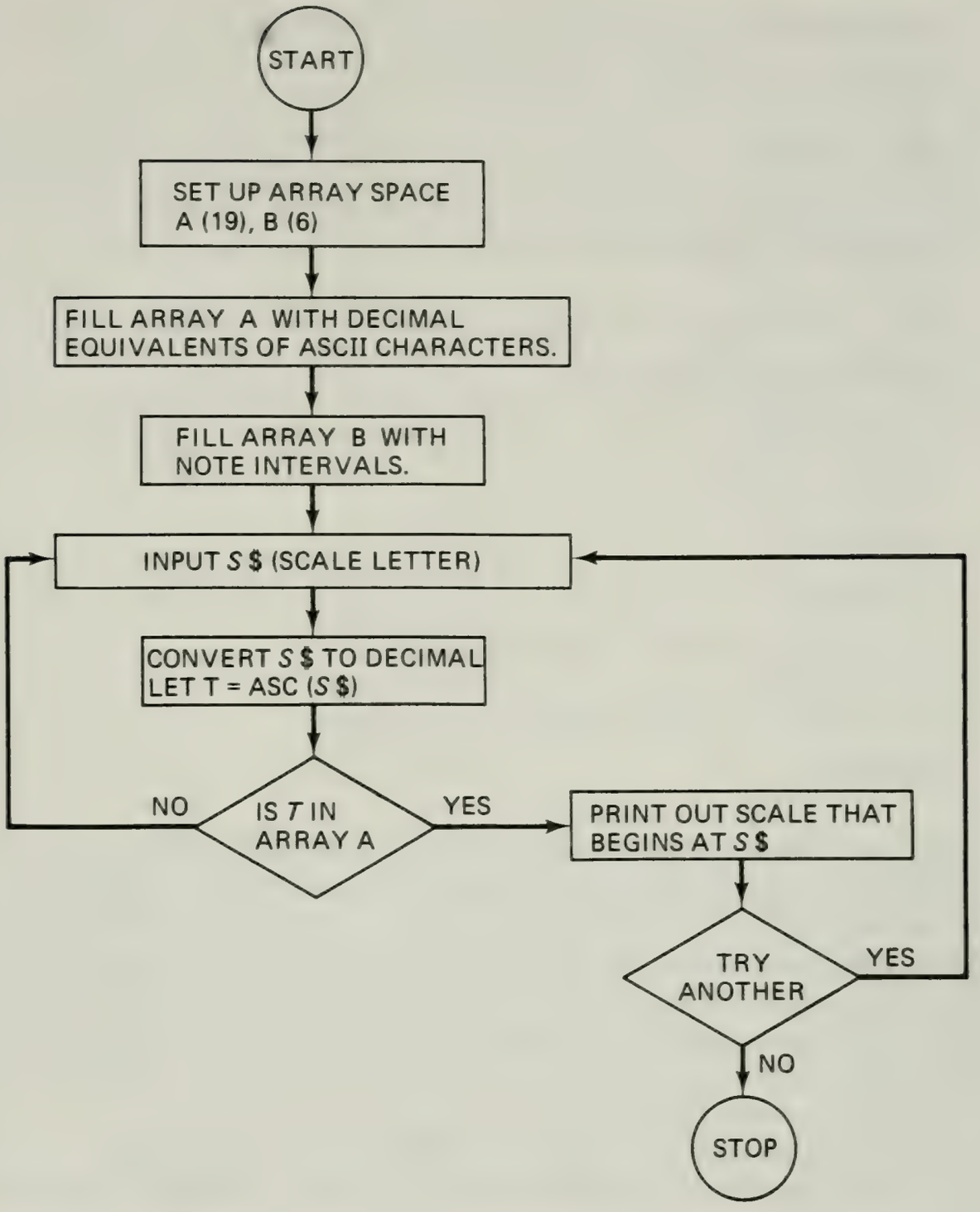


FIG. 4-9. Flowchart for music scale spelling program.

```
240 LET T2=ASC(T2$)
250 LET T=T1+T2/100
260 FOR I=1 TO 19
270 IF A(I)=T THEN GOTO 310
280 NEXT I
290 PRINT "SORRY, NO SUCH SCALE. TRY AGAIN":PAUSE 10
300 PRINT "&K"; GOTO 160
310 CURSOR 5,0
320 PRINT S$
```

```
330 CURSOR 5,3
340 FOR J=1 TO 6
350 LET K=3*J+3
360 LET N=B(J)
370 LET I=I+N
380 IF I>19 THEN LET I=I-19
390 LET T=A(I)
400 LET T1=INT(I)
410 LET T2=(T-T1)*100
420 SET DB=T1
430 SET DB=T2
440 CURSOR 5,K
450 NEXT J
460 PRINT S$
470 PRINT "ANOTHER SCALE? Y OR N"
480 INPUT B$
490 IF B$="Y" THEN PRINT "&K": GOTO 160
500 END
```

We associate the C-major scale with *do-re-mi-fa-sol-la-ti-do,* and it is "spelled" C-D-E-F-G-A-B-C. It is composed of seven notes, with one C on the bottom and another an octave higher on the top. Similarly, the G-major scale, for example, is spelled G-A-B-C-D-E-F#-G. The symbol F# stands for F sharp, and it represents a note which is one half-tone higher than the F appearing in the C-major scale. By the same token, there are notes which sound one half-tone lower than their root notes and they are called *flats,* marked by a *b.*

One curious quality about the spelling of musical scales is that some notes have dual identities. Thus we see, for example, in Fig. 4-10

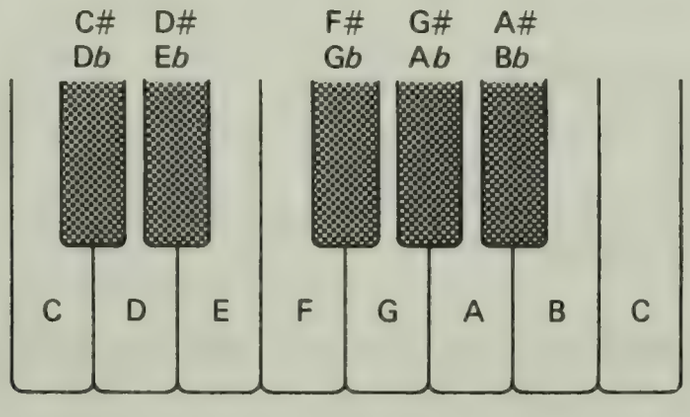


FIG. 4-10. The C-major scale as represented on the white keys of a piano. Some notes on the piano keyboard may take on variable identities, such as D# / E*b*.

that C# may also be considered D*b*. From a music theory viewpoint, the distinction is an important one, even though the scales C#-D#-E#-F#-G#-A#-B#-C# and Db-E*b*-F-G*b*-A*b*-B*b*-C-D*b* are the *same notes* when played on the piano. The two scales function differently, depending upon the key in a composition and may lead to entirely different tonal conclusions. A computer program which proposes to teach the spelling of musical scales must therefore take such nuances into account.

Another property of the musical scale is that the notes of a scale are not all equally spaced one from another. If we consider each key on the piano, whether black or white, to be half a step away from its neighbor, then we might consider a major scale to be laid out as follows:

WHOLE STEP
WHOLE STEP
HALF STEP
WHOLE STEP
WHOLE STEP
WHOLE STEP
HALF STEP

This pattern is true for any major scale, no matter which key it begins on; and a successful music instruction program must also address this property.

Consider Fig. 4-11. It depicts all of the notes contained in one octave, including notes which have more than one identity, arrayed in a circle. We can see that by starting at any point on this circle and them following a clockwise path according to the whole-step-half-step pattern we can spell any of the major scales.

With the foregoing information as background we can now return to the program listing to determine more precisely how the program works. We begin with the data statements:

```
20 DATA 67.32,67.35,68.98,68.32,68.35,69.98
30 DATA 69.32,69.35,70.32,70.35,71.98,71.32
40 DATA 71.35,65.98,65.32,65.35,66.98,66.32,66.35
50 DATA 3,3,2,3,3,3
```

Lines 20, 30, and 40 are the decoded notes, while line 50 is the whole-step-half-step indexing pattern. The note values (which are later transferred into array A in lines 60 to 80) are the decimal equivalents of the ASCII alphabet, as follows:

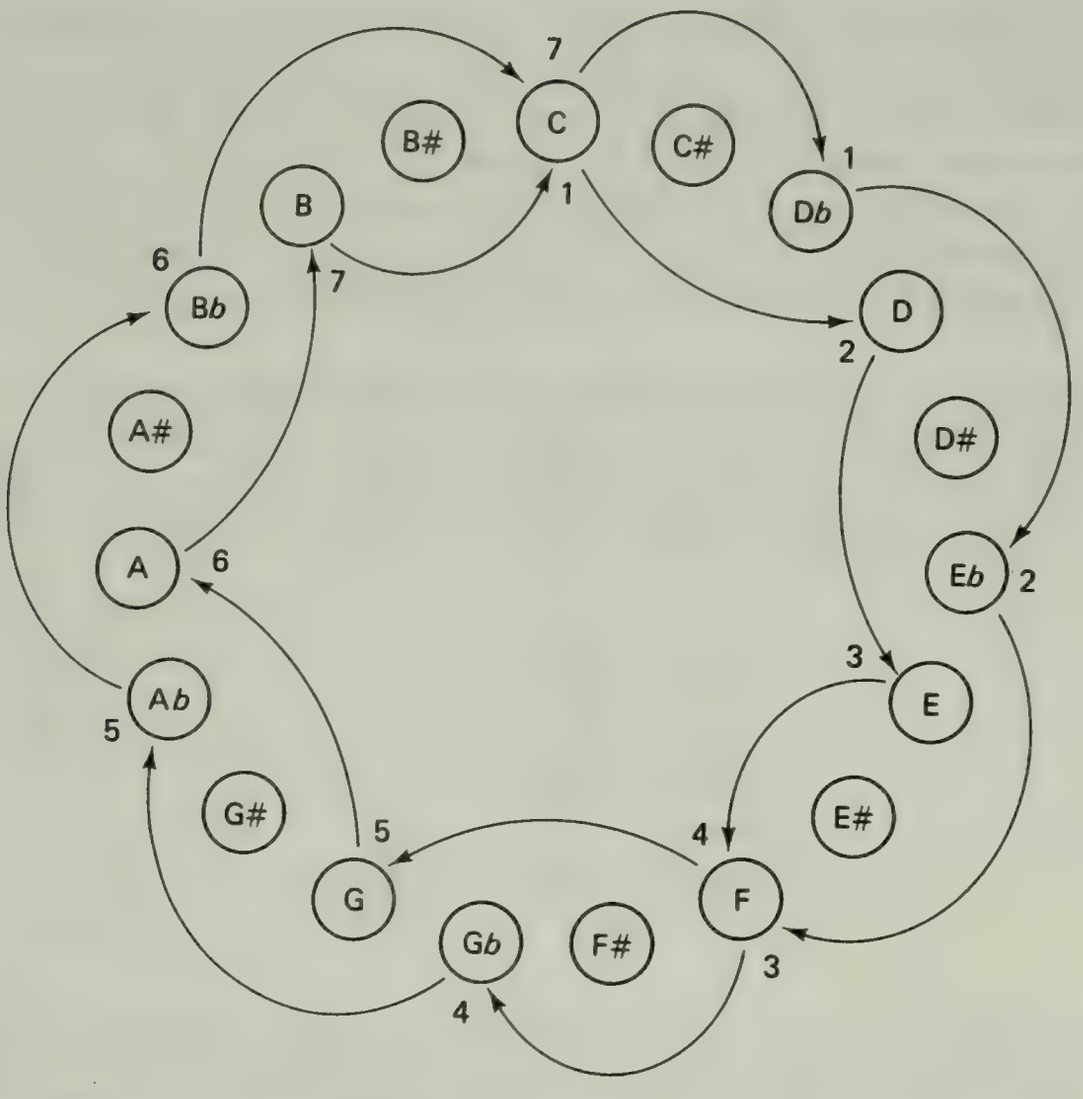


FIG. 4-11. One octave of notes on a piano keyboard arranged in a circle. By beginning at any note and travelling in a clockwise direction, any scale may be spelled out. In this diagram a whole step skips two notes, and a half step skips one note.

A=65
B=66
C=67
D=68
E=69
F=70
G=71

In addition, to provide for all possibilities, all notes of the musical scale are considered in this program to consist of two characters and are therefore represented as a two-digit whole number followed by a two-digit fraction:

A#=65.35
D♭=68.98
F=70.32

according to the scheme

= .35
♭ = .98
blank space = .32

Whenever a letter is entered from the keyboard it is first converted to its decimal equivalent. It is then compared against all of the entries in array A to determine if it is a valid musical note:

```
160 INPUT=S$
170 LET S=LEN(S$)
180 IF S=2 THEN GOTO 210
190 LET T=ASC(S$)+.32
200 GOTO 260
210 LET T1$=S$(1)
220 LET T2$=S$(2)
230 LET T1=ASC(T1$)
240 LET T2=ASC(T2$)
250 LET T=T1+T2/100
260 FOR I=1 TO 19
270 IF A(I)=T THEN GOTO 310
280 NEXT I
290 PRINT "SORRY, NO SUCH SCALE. TRY AGAIN":PAUSE 10
```

We know from previous experience that when the computer accepts string-variable input from the keyboard it ignores trailing blanks. If we asked for the scale spelling of the key of D, for instance, S$ would be equal to D at line 160 which, if converted directly into decimal, would be 68. Certainly D is a valid note, but searching array A for its decimal equivalent of 68 would yield no match and would result in an error message (line 290). Therefore, line 190 takes all single-character entries and appends the decimal equivalent of a blank space onto them.

If an entry consists of two characters such as B♭, the block of code from lines 210 to 250 splits the two characters apart, separately converts each to its decimal value, and then recombines them into the four-digit number described earlier.

If this newly constructed number matches one of the numbers in array A, the computer proceeds to spell out the musical scale which begins with that note. Referring to Fig. 4-11, we realize that arranging the notes in a circle is a useful device for humans; but it is worthless to the computer, since data arrays in BASIC are linear, not circular. Suppose, for example, that we began at the top of the circle, at C, and counted clockwise around the circle. If we counted C as 1, then by the time we reached B# we would be at 19. Suppose we were to continue counting around the circle. 20 would bring us back to C, 21 would be C#, and 38 would be two revolutions around the circle, pointing again at B#. In BASIC, however, if we reference A(19) we are pointing to B#, but if we reference A(21) we are pointing out-of-bounds—and execution halts on a fatal error.

The program gets around this difficulty by allowing us to conceptualize the notes in a circular array, while in actuality they are stored in an offset indexing and compensating scheme to stay within the confines of a 19-element array. For example, if we ask for the D-major scale, it converts the D of S$ into 68.32. Upon searching through array A the program finds that 68.32 occupies position 4 of the array; and using this value as its offset, it pulls consecutive increments from array B and adds them in sequence to the offset. In this way the program arrives at an index value which points to the next note in the scale of D-major.

We can play computer to see exactly how this works:

J (*master loop index*)	*I* (*old pointer*)	+	*B(J)* (*increment*)	=	*I* (*new pointer*)	*A(I)* (*note*)
1	4		3		7	E
2	7		3		10	F#
3	10		2		12	G
4	12		3		15	A
5	15		3		18	B
6	18		3		2	C#

We can see how linear array A is made to act as though it were circular. If *I*, the index variable that points to the notes in array A, becomes greater than 19 (the number of elements in array A) then *I* is reduced by 19. This has the effect of resetting the index pointer to the beginning of the array. The letter which names the scale, called the root (in this case D) is printed out separately before and after the loop indexed on *J* is executed, yielding:

D-E-F#-G-A-B-C#-D

Suppose we run the program:

```
RUN
WELCOME TO LEARN-A-SCALE
REPRESENT SHARPS WITH A SHIFTED 3 (#)
AND FLATS WITH A LOWER CASE B (b)
WHAT SCALE WOULD YOU LIKE TO SEE?
?C
C D E F G A B C
ANOTHER SCALE? Y or N Y
?G
G A B C D E F# G
ANOTHER SCALE? Y OR N Y
?Ab
Ab Bb C Db Eb F G Ab
ANOTHER SCALE? Y OR N Y
?Gb
Gb Ab Bb B# Db Eb F Gb
```

Even if we were not musicians we should be able to recognize that the *Gb* scale printed out by the computer is incorrect; intuition alone should tell us that no letter should appear twice in an alphabet. In the *Gb* scale just presented, however, B appears twice, first as B*b* then as B#; they are not the same note but they are the same letter, and that is incorrect.

A music textbook would list for us all of the major scales:

```
C  D  E F  G  A  B  C
   D  E F# G  A  B  C# D
      E F# G# A  B  C# D# E
        G  A  B  C  D  E  F# G
           A  B  C# D  E  F# G# A
              B  C# D# E  F# G# A# B
                 C# D# E# F# G# A# B# C#
                    F# G# A# B  C# D# E# F#
Db Eb F Gb Ab Bb C  Db
   Eb F G  Ab Bb C  D  Eb
      F G  A  Bb C  D  E  F
```

```
Gb Ab  Bb Cb  Db  Eb  F   Gb
   Ab  Bb C   Db  Eb  F   G   Ab
       Bb C   D   Eb  F   G   A   Bb
          Cb  Db  Eb  Fb  Gb  Ab  Bb  Cb
```

Before we were to release this program as a valid music instruction aid we would first have to test its ability to correctly reproduce each one of the 15 major scales just listed. In doing so we would discover that the computer performs correctly on all but just two of the scales: G*b*-major and C*b*-major. We have just seen the results of a request to write out the G*b* scale; asking for the C*b* scale leads to something else entirely:

```
WHAT SCALE WOULD YOU LIKE TO SEE?
?Cb
SORRY, NO SUCH SCALE. TRY AGAIN
```

If there is not a C*b* in the program, there should be, since it is listed as one of the 15 major scales. Referring again to Fig. 4-11 we see that C*b* is in fact not on the graph, and closer inspection tells us that another note in the C*b* scale, F*b*, is also not included.

Suppose we try to correct this oversight by adding C*b* and F*b* to the circle of notes, as in Fig. 4-12. Now absolutely all of the notes in an octave are represented. The diagram on the inside of the circle of notes shows what happens when we trace out the scale of C on the circle: all of the letters of the scale are now equally spaced apart from each other around the circle.

If we use the same spacing scheme and try to trace out the C*b* scale we get:

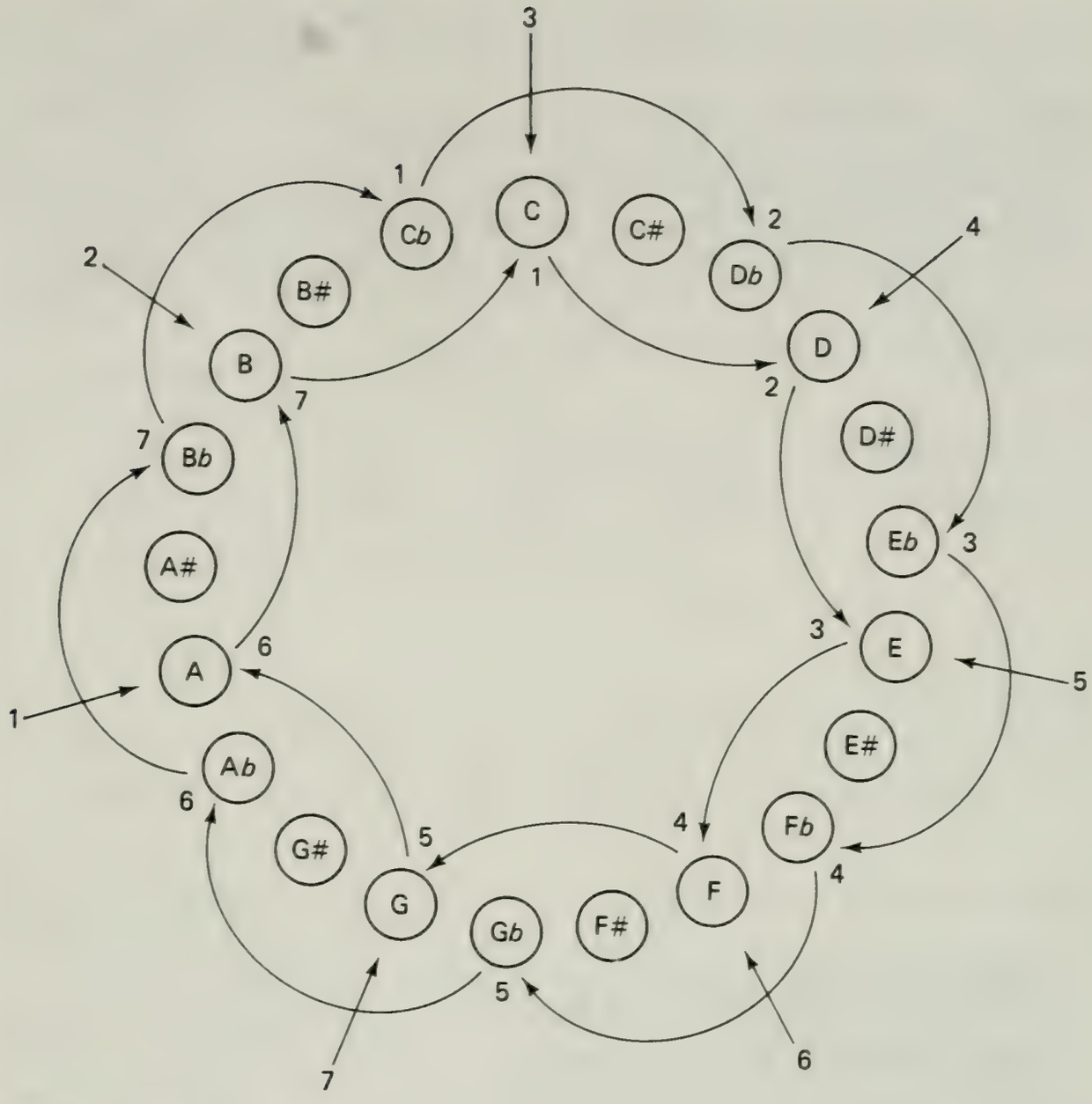
C*b*–D*b*–E*b*–F*b*–G*b*–A*b*–B*b*–C*b*,

which we can check against our reference book to establish its correctness. Suppose we use the same spacing scheme to trace out the scale of A, shown in Fig. 9-12 by the arrows. This yields:

A-B-C-D-E-F-G-A

which is incorrect.

Indeed, if we use the listing of the 15 major scales as a guide, and trace out the various scales on the revised circle of notes we would discover the following:



Inside: C D E F G A B C

Outside: C*b* D*b* E*b* F*b* G*b* A*b* B*b* C*b*

Arrows: A B C D E F G A

FIG. 4-12. Revised circle of notes, including all sharps and flats.

Name of Scale	*Note Pattern Around Circle of Notes*
C	3-3-3-3-3-3-3
D	3-4-2-3-3-4-2
E	4-3-2-3-4-3-2
G	3-3-3-3-3-4-2
A	3-4-2-3-4-3-2
B	4-3-2-4-3-3-2

If we continued we would see that some of the patterns repeat; but clearly, if we were to use this method in our music instruction routine the program would require some extensive rewriting, and the result would not be nearly as clean as it is in its present state.

The best solution in this case, then, is to put in a patch and leave the rest of the program as it stands. The patch would only have to recognize the two special cases, *Gb* and *Cb*; if either of those two notes were requested, the patch would skip all of the processing in the main body of the program and simply "brute force" an answer:

```
161 IF S$="Gb" THEN GOTO 462
162 IF S$="Cb" THEN GOTO 464
```

—

—

—

```
461 GOTO 470
462 PRINT "Gb Ab Bb Cb Db Eb F Gb"
463 GOTO 470
464 PRINT "Cb Db Eb Fb Gb Ab Bb Cb"
```

Before we leave the music instruction program we must attend to one other loose end. Our listing of major scales has only 15 entries, yet our circle of notes (the original version) numbers 19. If we further eliminate the two scales *Cb* and *Gb*, which we arrive at via brute force, it would seem to leave only 13 scales out of a potential 19. Where are the other six scales? If we ask for the scale of A#, for example, we would see:

```
WHAT SCALE WOULD YOU LIKE TO SEE?
?A#
A# B# Db D# E# Gb Ab A#
```

Actually, there is no such thing as an A#-major scale. If there were, it would have to be written thus:

A# B# C## D# E# F## G## A#

Because this scale must be written with double-sharped notes, it is not one of the 15 natural scales. Besides A#, rounding out the list of nonexistent major scales are B#, D#, E#, G#, and F*b*. We might therefore want to include one additional patch which would recognize these impossible scales and take the appropriate action:

```
163 IF S$="B#" THEN GOTO 290
164 IF S$="D#" THEN GOTO 290
165 IF S$="E#" THEN GOTO 290
166 IF S$="G#" THEN GOTO 290
167 IF S$="Fb" THEN GOTO 290
—

—
290 PRINT "SORRY, NO SUCH SCALE. TRY AGAIN":PAUSE 10
```

With this final patch the music instruction program would be essentially debugged and ready to handle any situation.

We have seen in this chapter how patches are used to rescue a floundering program by forcing the code to handle specific situations which were not provided for by the original algorithm. Even with careful planning it often is very difficult for a programmer to see the problem from all possible sides and to provide for all possible applications to which the program must be put. Patches are therefore often used as a quick-and-dirty method of fixing up a program in the face of such unforeseen circumstances.

5

Using Debugging Tools

In the previous chapters we have examined a number of techniques for preventing, finding, and correcting errors (bugs) in a computer program; in this chapter we will integrate all that we have learned into a complete programming effort.

We begin with the specification:

> The program will simulate the ecological and environmental interactions between two competing groups. One group will be herbivorous (plant eating) while the other group will be carnivorous (meat eating). Group 1 will be prey to group 2. The program will provide tabular output describing group population sizes as a function of time.

We can now make up a list to amplify those aspects of the problem with which we feel the computer program might concern itself. We have been asked to construct an ecological/environmental model with the computer, and then to use that model to diagram and predict the results of a two-species interaction. The specification lists for us the three main elements of our model:

1. Quantity of plant life
2. Amount of species 1
3. Amount of species 2

We can expand each of these three main headings by listing below them the factors that affect and influence them:

1. Quantity of plant life
   1) Amount of land available to plants
   2) Quality of soil
   3) Amount and pattern of rainfall
   4) Amount and pattern of sunlight
   5) Number of destructive insects
2. Amount of species 1 (herbivores)
   1) Quantity of plant life
   2) Hunting pressure from animal group 2
   3) Birthrate
   4) Average natural lifespan
   5) Frequency and severity of disease
   6) Natural disaster (fire, flood, etc.)
3. Amount of species 2 (carnivores)
   1) Amount of animal group 1
   2) Birthrate
   3) Average natural lifespan
   4) Frequency and severity of disease
   5) Natural disaster (fire, flood, etc.)

Many of these subheadings are, in turn, influenced by additional factors, as well as by each other. For example:

Amount of land available to plants
is influenced by

Amount of land occupied by animals,
which in turn is influenced by

Total number of animals,
which is in part dependent upon

Amount of species 1
which, among other things, is influenced by

Quantity of plant life
which is a function of

Amount of land available to plants.

Nothing in an ecological system functions with total independence: eliminating all of the pests in a section of cropland, for instance, might set off a disastrous chain reaction involving the birds which feed on those bugs, the mammals which feed on the birds, the parasites which depend upon the mammals, and so on.

For completeness, then, we would like our program to take into consideration all possible (or at least all relevant) interacting and competing influences. Unfortunately, such is not possible: not all relevant forces in any given ecological system are known, and for those which are known, scientists have an imperfect understanding of how they relate and even on how many levels they interact.

We will therefore have to make a number of simplifying assumptions when we construct our computer model of the two-species ecosystem. One way to do this is to separate the various parameters into two classes: those which are dependent upon each other, and those which are independent.

Thus we might say that the amount of species 2 would be dependent upon a number of factors, one of which is that species' birth rate. Yet for the purposes of our computer simulation, the birthrate will be established externally and will be independent of any influences from the program. This introduces inescapable artificiality into the simulation, which means that if the results are to be at all valid the simplifying assumptions which we make must at least be defensible.

In a way we are patching the program before it has even been written. We know from the outset that certain aspects of the program would be impossible to program correctly (with respect to the real world) because we do not know enough about the processes in nature to be able to accurately describe them for the computer. So we patch or brute-force that part of the problem based on our best guess as to what a reasonable value might be.

Suppose, therefore, that we classify the dependent and independent variables as follows:

Dependent Variables

Quantity of plant life
- Amount of land available to plants
- Grazing pressure from animal group 1

Amount of species 1
- Hunting pressure from animal group 2

Amount of species 2

Independent Variables

Plant life yield per unit area of land
Birth rate, group 1
Grazing pressure from group 1 (plants/animals)
Birth rate, group 2
Hunting pressure from group 2 (herbivores/carnivores)

In addition, we will want to classify some events as "chance" and will represent them in the program as being dependent upon some random-occurrence generator. Such "chance" items would include:

Frequency and severity of disease
Natural disaster (fire, flood, etc.)

By comparing this revised list with the original list, we can see that a number of factors which we at first determined to be significant have been left out:

Quality of soil
Amount and pattern of rainfall and sunlight
Number of destructive insects
Average natural lifespan for groups 1 and 2

These items have been cut from the program in the interest of simplification. Ideally, the assumptions we make when setting the values of the independent variables and when coding the mathematical relationships of the dependent variables will take into account these factors on the general level, at least, compensating for their not having been explicitly included.

Having discussed the preliminaries of the task we might return to the original specification and modify it to reflect our assessment of what can and cannot be reasonably done in a modest computer simulation. Thus we add to the original specification:

For the purpose of the simulation, environmental factors will be divided into interdependent, independent, and chance.

We would then list the factors which we had classified earlier.

Normally our next step would be to lay out a flowchart. For a simulation program, however, before we can arrive at any decisions about the order of statement we must plot out the interactions of the various subsections of code, as in Fig. 5-1. This figure clearly illustrates the dis-

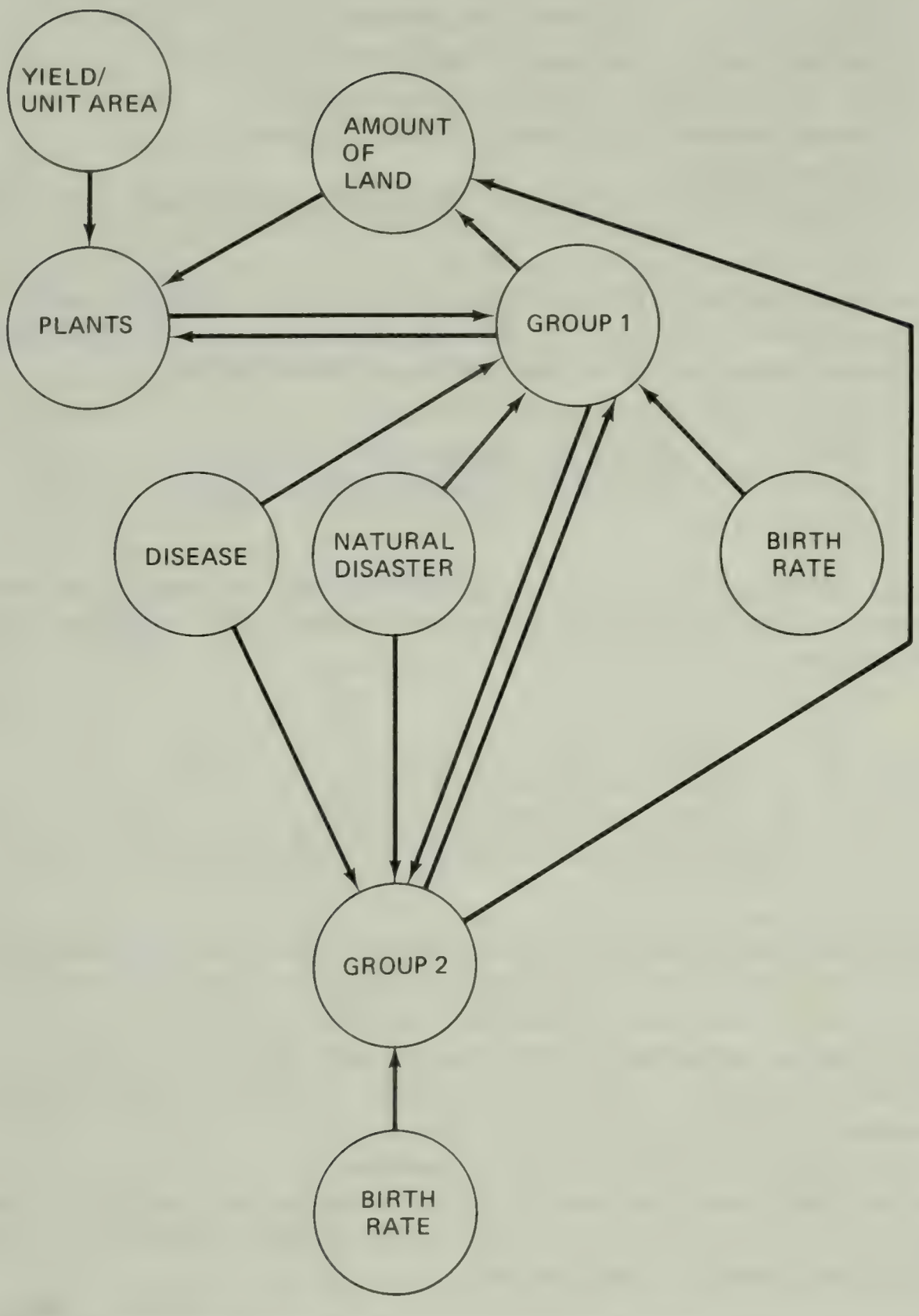


FIG. 5-1. Interaction diagram for the ecosystem simulation program.

tinction between dependent and independent elements: dependent elements have arrows flowing both towards and away from themselves, while independent elements have arrows which only flow away.

Suppose from the block diagram in Fig. 5-1 we generate the flowchart shown in Fig. 5-2. Note that we have identified the two groups in

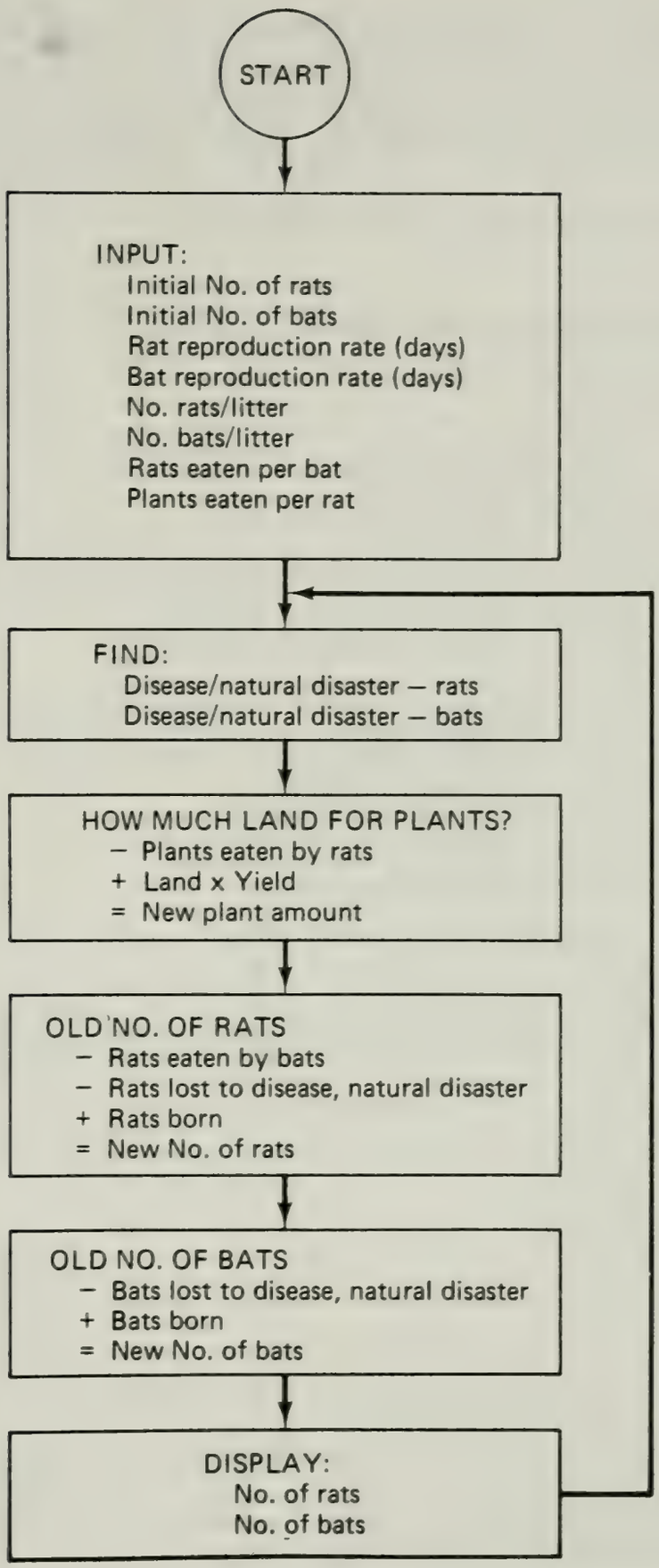


FIG. 5-2. Flowchart for two-species ecosystem model (Moon Rats vs. Space Bats).

our simulation as moon rats and space bats. We could have used two more realistic groups such as rabbits and foxes, but such a choice would invalidate the design intention of the simulation program: to write a *general* program which simulates the gross interaction of a two-group system.

The resulting code might be as follows:

```
10 REM SIMULATION PROGRAM
20 REM SPACE BATS VS MOON RATS
30 LET L=10000,I=1,J=1,K=1
40 PRINT "TWO GROUP ECOSYSTEM SIMULATION"
50 PRINT "SPACE BATS VS MOON RATS"
60 PRINT "BATS EAT RATS AND RATS EAT PLANTS"
70 INPUT "INITIAL POPULATION OF RATS",R
80 INPUT "INITIAL POPULATION OF BATS",B
90 INPUT "RAT REPRODUCTION RATE(DAYS)",R1
100 INPUT "BAT REPRODUCTION RATE(DAYS)",B1
110 INPUT "NO. OF RATS/LITTER",R2
120 INPUT "NO. OF BATS/LITTER",B2
130 INPUT "RATS EATEN/BAT",R3
140 INPUT "PLANTS EATEN/RAT",P2
150 LET X=INT(10*RND(0) )
160 ON X GOTO 280,170,280,190,280,210,280,230,280
170 LET D=.1
180 GOTO 240
190 LET D=.2
200 GOTO 240
210 LET D=.3
220 GOTO 240
230 LET D=.4
240 LET X1=INT(10*RND(0) )/2
250 IF(10*X1)-(INT(X1) )*10=0 THEN LET D1=1 ELSE
LET D1=2
260 GOTO 290
270 REM NO DISASTER
280 LET D=0
290 LET L=L-(R*3.58E-8)-(B*1.07E-7)
300 LET P=P+(L*4E5)-(R*P2)
310 LET R=R-(B*R3)
```

```
320 IF I=R1 THEN LET I=0:LET R=R+(R*R2)
330 IF D1=1 THEN LET R=R-(R*D)
340 IF J=B1 THEN LET J=0:LET B=B+(B*B2)
350 IF D1=2 THEN LET B=B-(B*D)
360 PRINT "DAY, RATS VS BATS";K;R;B
370 LET I=I+1
380 LET J=J+1
390 LET K=K+1
400 GOTO 150
410 END
```

Before we can make any serious attempt to test this program we must first understand how it is written. We note, for example, that only eight of the 17 variables are described by internal documentation:

Variable	*Description*
R	Population of rats
B	Population of bats
R1	Rat reproduction rate (days)
B1	Bat reproduction rate (days)
R2	No. rats/litter
B2	No. bats/litter
R3	No. rats eaten/bat
P2	No. plants eaten/rat
X	?
D	?
X1	?
D1	?
L	?
P	?
I	?
J	?
K	?

If this simulation program belonged to someone else, and we came upon it "cold" and were asked to help debug it, we would likely find it very difficult to provide descriptions for the undocumented variables. The variable X, for example, seems to have something to do with a random number:

```
150 LET X=INT(10*RND(0) )
```

Once selected, X is used in a "computed GOTO" statement:

```
160 ON X GOTO 280,170,280,190,280,210,280,230,280
```

The flow of execution is switched down one of five different paths, depending upon the value of X. The difference in each path is the value to which the variable *D* is set, but since we also do not know the definition of variable *D*, the significance of the five paths is unclear.

Suppose we insert the following lines of documentation:

```
32 REM L=TOTAL LAND AREA AVAILABLE (SQ MILES)
34 REM I=RAT REPRODUCTION CYCLE COUNTER
36 REM J=BAT REPRODUCTION CYCLE COUNTER
38 REM K=INCREMENTING DAY COUNTER
155 REM X=RANDOM DETERMINATE FOR NATURAL DISASTERS
165 REM D=NATURAL DISASTER SEVERITY
245 REM X1=RANDOM SELECTION OF RATS OR BATS
255 REM D1=IF X1 IS EVEN, D1=1,RATS FEEL DISASTER
257 REM       IF X1 IS ODD, D1=2,BATS FEEL DISASTER
305 REM P=NO. OF PLANTS
```

Suppose further that we "verbalize" some of the equations. This technique is not often used, but in simulation or modeling programs, especially, it helps us as we try to place a real-world handle on the structured language of the mathematical statements. Thus

```
290 LET L=L-(R*3.58E-8)-(B*1.07E-7)
```

translates as: let the new quantity of land area available to the plants be equal to the old land area, minus the total area of land occupied by the rats and the bats put together. We also see in this equation two of the simplifying assumptions which we are forced to make: when computing the land area lost to rats and bats respectively, we assume 3.58×10^{-8} square miles, or one square foot for each rat, and three square feet for each bat.

The next line of code illustrates more assumptions:

```
300 LET P=P+(L*4E5)-(R*P2)
```

Variable *P*, as we now know from the remark at line 305, refers to the

number of plants. The equation therefore translates as: Let the new quantity of plants be equal to the old quantity of plants, plus the quantity produced by the currently available area of land, and minus the quantity of plants consumed by the moon rats. The simplifying assumption here is that each square mile of land produces 400,000 plants, a number which we hope on the average takes into account such factors as amount of rainfall, amount of sunlight, length of growing season, fertility of soil, efficiency of natural pollination, and so forth. The amount of plants consumed by the total rat population is controlled by two variables: one dependent, which is the current size of the rat population, and one independent, which is the number of plants consumed in one cycle by each rat (a quantity input from the keyboard at line 140).

Lines 310, 320, and 330 compute the updated number of moon rats for this cycle:

```
310 LET R=R-(B*R3)
320 IF I=R1 THEN LET I=0: LET R=R+(R*R2)
330 IF D1=1 THEN LET R=R-(R*D)
```

Line 310 accounts for losses incurred due to hunting of moon rats by space bats. Variable *B*, the number of bats, is a dependent variable; *R*3, the number of rats eaten per bat per cycle, is independent.

The next line accounts for additions to the rat population due to new births and illustrates both a crucial decision in our construction of the ecology model and a simplifying assumption of questionable validity. The first aspect concerns the placement of the equation *after* the point where the rats' hunting losses are totaled up: it is conceivable that predation by the space bats might decimate the moon rat population before the gains from new births are added in, making recovery for the rat population impossible. Also, in one of our most serious simplifications we have assumed that the rat population reproduces only every *R*1 number of cycles, and that at that time every member of the population produces *R*2 number of offspring. The reality of the situation is that *every* cycle a certain percentage of the rat population is reproducing and, further, that this percentage changes in response to a number of influences each cycle. Only a test run of the program will tell us if our simplification is workable. We may yet find that a more detailed mathematical treatment of moon rat population dynamics is required if we are to have anything approaching reasonable results.

Line 330 addresses the problem of random events which might affect the populations of the two test groups (in the present case we are dealing with just one group, the rats). Variable *D*1 is the variable which determines whether the chance event will affect the rats or the bats. If *D*1 equals 1 the rats suffer, if it equals 2 the bats' numbers are affected.

The variable *D* determines the severity of the chance event. Both *D* and *D*1 are computed in the block of code running from line 150 to 280:

```
150 LET X=INT(10*RND(0) )
160 ON X GOTO 280,170,280,190,280,210,280,230,280
170 LET D=.1
180 GOTO 240
190 LET D=.2
200 GOTO 240
210 LET D=.3
220 GOTO 240
230 LET D=.4
240 LET X1=INT(10*RND(0) )/2
250 IF (10*X1)-(INT(X1) )*10=0 THEN LET D1=1 ELSE
LET D1=2
260 GOTO 290
270 REM NO DISASTER
280 LET D=0
```

D, as used in line 330, is a multiplier, and we see *D* being set to 0.1, 0.2, 0.3, 0.4, or zero at lines 170, 190, 210, 230, and 280, depending upon the value of X. Variable X, meanwhile, is a random number which fluctuates from zero to nine. The five choices for *D* is another example of a preprogrammed patch. In the SOL version of BASIC, the RND(0) function returns all random numbers less than one (1). Suppose, then, that we had done the following:

```
150 LET D=RND(0)
```

On the surface it looks as though this one statement would do the job of at least 10 statements in the original version of the program; each time the statement would be executed it would produce a decimal fraction which could be used directly as the chance event severity coefficient.

Indeed, we may find that at some future time we may want to replace the original version with this one-line modification, but for the present we want to retain as much control over the program as possible. With the original version we can force the chance events variable into only one of five discrete values, rather than allow it to assume one of the many thousands of values generated by the RND function. Also, we have insured that the maximum amount of rats or bats lost to chance catastrophe will be limited to no more than 40%.

Before we move on to translating the mathematical sentences which describe the behavior of the bat population, we can see that, thanks to the documentation newly included in the program, we have discovered yet another error:

```
310 LET R=R-(B*R3)
320 IF I=R1 THEN LET I=0:LET R=R+(R*R2)
330 IF D1=1 THEN LET R=R-(R*D)
```

These three lines are the only lines which describe the changes that occur each cycle to the rat population, yet when we associate all of the variables with their definitions we find that nowhere does *P*, the number of plants, enter in. However, our group interaction diagram (Fig. 5-1) shows the number of plants as having a very definite effect upon the number of rats: if there are more rats than the plant population can support, the surplus rats can be expected to starve from lack of food.

Again we must make a simplifying assumption. In reality, if population outdistanced the food supply a general famine would result with only a small percentage of the population dying the first year (cycle), followed by a significantly increased percentage dying in the next year and years after if the famine persisted. However, we are already tied into the simplification that chance events which *do* occur last for only one cycle, and at no time do such events remove more than 40% of the population.

Suppose we adopt the following simplification in regard to the rat population/plant quantity balance:

```
IF R>P/P2 THEN LET R=.9*(P/P2)
```

We are saying that if the rat population is greater than the number of rats which the plant population can support (*P*/*P*2), then reduce the rat population to 9/10 of the maximum support number.

Having discovered a potential bug in the program, we have coded what we hope is a reasonable patch, but where in the program do we put it? Should it be inserted before line 310 where the rat population is adjusted to predation by bats, or after 310 and before 320 where the rat population is increased in response to the birthrate? Or perhaps after 320 and before 330, where the population decreases as a result of random natural disaster; or should it go at the end of all three lines?

Each suggested placement implies different interpretations of our understanding of the ecosystem we are trying to model; thus, each placement may lead to a significantly different outcome. Suppose we insert the patch between lines 310 and 320:

```
315 IF R>P/P2 THEN LET R=.9*(P/P2)
```

The computation of the bat population is handled similarly, except that bats are not subject to predation:

```
340 IF J=B1 THEN LET J=0:LET B=B+(B*B2)
350 IF D1=2 THEN LET B=B-(B*D)
```

In this case we are first increasing the bat population if the reproduction rate index matches the value supplied at the beginning of the simulation. We note that, as with the rats, we have made the simplifying assumption that every *B1* number of cycles every member of the current bat population produces *B2* number of offspring.

Line 350 refers to bat population losses resulting from chance natural disasters.

Comparing the code with the interaction diagram of Fig. 5-1, we notice that once again we have failed to include a necessary factor: the bat population is influenced by the size of the rat population in much the same way that the rats were influenced by the number of plants, yet the coded equations do not reflect this.

Realizing the implications of proper equation construction and placement, suppose we introduce the following statement:

```
335 IF B>R/R3 THEN LET B=.9*(R/R3)
```

All computer modeling schemes involve simplification, approximation, and in some cases, informed guesswork. This is especially true in our ecosystem simulation of moon rats vs. space bats; our goal is to illustrate general two-species interaction but in doing so we run the risk of simplifying ourselves to the level of nonsense. At the very least, we can expect a first trial execution of the program to exhibit a number of bugs.

Therefore, to save ourselves time and trouble later, we will now apply the techniques of earlier chapters and design in some debug snapshots which we can activate when the need arises.

Structurally, the program is short enough and simple enough that one snapshot routine will suffice; we have listed all of the pertinent variables but a short time ago. Quite the opposite of a tricky programming task, then, our snapshot routine will mainly be an exercise in clean, straightforward organization and display of diagnostic information.

One such design might be:

```
1000 REM SIMULATION SNAPSHOT
1010 REM ORGANIZE AND PRINTOUT
```

```
1020 REM ALL RELEVANT DATA
1030 PRINT "&K"
1040 PRINT "SNAPSHOT CALLED FROM LINE";Y
1050 PRINT "DAY=";K;"RAT INDEX=";I;"BAT INDEX=";J
1060 PRINT "  ATS REPRODUCE EVERY";R1;"DAYS, YIELDING";
R2;"YOUNG"
1070 PRINT "BATS REPRODUCE EVERY";B1;"DAYS, YIELDING";
B2;"YOUNG"
1080 PRINT "BATS EAT";R3;"RATS, RATS EAT";P2;"PLANTS"
1090 PRINT "LAND FOR PLANTS=";L;"NO. PLANTS=";P
1100 PRINT "DISASTER SEVERITY=";D;"SELECTOR CODE=";D1
1110 PRINT "X=";X;"X1=";X1
1120 PRINT "NO. OF RATS=";R;"NO. OF BATS=";B
1130 INPUT Z$
1140 RETURN
```

Recalling the advice from earlier chapters regarding the placement of print statements and incorporating the changes already mentioned so far for this program, the revised listing looks as follows:

```
10 REM SIMULATION PROGRAM
20 REM SPACE BATS VS MOON RATS
30 LET L=10000,I=1,J=1,K=1
32 REM L=TOTAL LAND AREA AVAILABLE (SQ MILES)
34 REM I=RAT REPRODUCTION CYCLE COUNTER
36 REM J=BAT REPRODUCTION CYCLE COUNTER
38 REM K=INCREMENTING DAY COUNTER
40 PRINT "TWO GROUP ECOSYSTEM SIMULATION"
50 PRINT "SPACE BATS VS MOON RATS"
60 PRINT "BATS EAT RATS AND RATS EAT PLANTS"
70 INPUT "INITIAL POPULATION OF RATS",R
80 INPUT "INITIAL POPULATION OF BATS",B
90 INPUT "RAT REPRODUCTION RATE(DAYS)",R1
100 INPUT "BAT REPRODUCTION RATE(DAYS)",B1
110 INPUT "NO. OF RATS/LITTER",R2
120 INPUT "NO. OF BATS/LITTER",B2
130 INPUT "RATS EATEN/BAT",R3
140 INPUT "PLANTS EATEN/RAT",P2
145 INPUT "IS THIS A DEBUG RUN?(RETURN FOR NO)",Z$
```

```
147 IF Z$="" THEN LET Z=0:GOTO 150 ELSE LET Z=1
150 LET X=INT(10*RND(0) )
155 REM X=RANDOM DETERMINATE FOR NATURAL DISASTERS
157 IF Z=1 THEN LET Y=150:GOSUB 1000
160 ON X GOTO 280,170,280,190,280,210,280,230,280
165 REM D=NATURAL DISASTER SEVERITY
170 LET D=.1
180 GOTO 240
190 LET D=.2
200 GOTO 240
210 LET D=.3
220 GOTO 240
230 LET B=.4
240 LET X1=INT(10*RND(0) )/2
245 REM X1=RANDOM SELECTION OF RATS OR BATS
250 IF (10*X1)-(INT(X1) )*10=0 THEN LET D1=1 ELSE
LET D1=2
255 REM D1=IF X1 IS EVEN, D1=1, RATS FEEL DISASTER
257 REM IF X1 IS ODD, D1=2, BATS FEEL DISASTER
259 IF Z=1 THEN LET Y=250:GOSUB 1000
260 GOTO 290
270 REM NO DISASTER
280 LET D=0
290 LET L=L-(R*3.58E-8)-(B*1.07E-7)
300 LET P=P+(L*4E5)-(R*P2)
305 REM P=NO. OF PLANTS
310 LET R=R-(B*R3)
312 IF Z=1 THEN LET Y=310:GOSUB 1000
315 IF R>P/P2 THEN LET R=.9*(P/P2)
320 IF I=R1 THEN LET I=0:LET R=R+(R*R2)
325 IF Z=1 THEN LET Y=320:GOSUB 1000
330 IF D1=1 THEN LET R=R-(R*D)
332 IF Z=1 THEN LET Y=330:GOSUB 1000
335 IF B>R/R3 THEN LET B=.9*(R/R3)
340 IF J=B1 THEN LET J=0:LET B=B+(B*B2)
345 IF Z=1 THEN LET Y=340:GOSUB 1000
350 IF D1=2 THEN LET B=B-(B*D)
355 IF Z=1 THEN LET Y=350:GOSUB 1000:GOTO 370
360 PRINT "DAY, RATS VS BATS";K;R;B
```

```
370 LET I=I+1
380 LET J=J+1
390 LET K=K+1
400 GOTO 150
410 END
```

We are now ready to give our program a test run:

```
RUN
TWO GROUP ECOSYSTEM SIMULATION
SPACE BATS VS MOON RATS
BATS EAT RATS AND RATS EAT PLANTS
INITIAL POPULATION OF RATS?
```

As we know from previous chapters, choosing meaningful input values when initially testing a program is something of an art as well as a science. Common sense is always a good place to start, however, and in the present example it tells us that, to begin with, we probably do not want to have more bats than rats, nor even an equal number of each. Also, rats should reproduce faster and more heavily than the predator bats.

We might therefore continue as follows:

```
INITIAL POPULATION OF RATS 100
INITIAL POPULATION OF BATS 10
RAT REPRODUCTION RATE(DAYS) 2
BAT REPRODUCTION RATE(DAYS) 4
NO. RATS/LITTER 10
NO. BATS/LITTER 5
RATS EATEN/BAT 6
PLANTS EATEN/RAT 4
IS THIS A DEBUG RUN?(RETURN FOR NO) ?
```

If we answer yes to the last question, we will generate the snapshot debugger output which is referenced seven times in the simulation program. This could result in considerable quantities of output, so we decide for the present to avoid resorting to our designed-in debuggers unless absolutely necessary.

This is not a copout. In general, if a program can be debugged using only the programmer's tools of understanding, insight, and deduction,

then it is only proper that it should be debugged that way. Among other things, such an approach weans the programmer away from an overdependence upon artifical aids and mechanical techniques. The more analytical brainwork a programmer invests in a particular routine, the less the likelihood of that routine exhibiting serious flaws.

We therefore hit a carriage return and the program continues:

```
DAY, RATS VS BATS 1      40      5.9999
BS ERROR IN LINE 240
```

Perhaps we made a typographical error when entering line 240, so we display the line and investigate:

```
240 LET X1=INT(10*RND(0) )/2
```

Nothing appears to be wrong with line 240. RND(0) returns a randomly selected number, we multiply that number by 10, take the integer portion of that result, and divide it by 2 to determine if it is odd or even.

There is one possibility of error, however: the RND function returns numbers between 1 and 0. Suppose it returned 0.3478; the equation would be evaluated as:

```
          RND(0) = 0.3478
       10*RND(0) = 3.478
  INT(10*RND(0) ) = 3
INT(10*RND(0) )/2 = 1.5
```

Suppose the value 0.0182 were returned:

```
          RND(0) = 0.0182
       10*RND(0) = 0.182
  INT(10*RND(0) ) = 0
INT(10*RND(0) )/2 = 0/2 = ?
```

Division into zero should equal zero, but perhaps in this particular sort of mathematical construction the computer becomes confused and sends an error message. An easy way to check is to subdivide line 240 into a couple of shorter commands while at the same time putting in a patch which excludes all zero values:

```
240 LET T=INT(10*RND(0) )
242 IF T=0 THEN GOTO 240
244 LET X1=T/2
```

We are ready to try again. The preliminary data input stage is the same as before, but the cycle-by-cycle listing has changed:

```
DAY, RATS VS BATS 1     40       4.799999
DAY, RATS VS BATS 2     123.20001     4.799999
DAY, RATS VS BATS 3     94.40001     3.8399999
DAY, RATS VS BATS 4     784.96012    23.04
DAY, RATS VS BATS 5     452.70408    23.04
DAY, RATS VS BATS 6     3459.1049    16.128
DAY, RATS VS BATS 7     2689.8695    16.128
DAY, RATS VS BATS 8     28525.117    96.768
DAY, RATS VS BATS 9     25149.158    96.768
DAY, RATS VS BATS 10    189177.83    96.768
```

Aside from the fact that this free-formatted output is messy and hard to read, and aside from the fact that there is no such thing as fractional rats or bats, it looks as though the program may be working. Both populations show a steady increase despite temporary setbacks due to natural disasters and predation.

The output form and the fractional rats and bats can be taken care of with a reworded print statement:

```
360 PRINT ''DAY, RATS VS BATS''; %4I;K;%12I;R;B
```

The "%4I" and "%12I" establish formatted integer fields that are 4 and 12 characters wide, respectively. Running the program now yields:

```
DAY, RATS VS BATS 1                    40
FO ERROR AT LINE 360
```

"FO" is a *field overflow* error, as explained in the SOL's Extended Cassette BASIC user's manual:

> An attempt has been made to print a number larger than extended BASIC's numerical field size.

Such an explanation hardly seems possible, since we know for a fact that the first time through the program the total bat population, which has not yet had a chance to be increased due to reproduction, cannot possibly be greater than 10. In an integer field of 12 characters we would not expect such a small number to cause an overflow.

If we refer to the user's manual for the precise description of the integer print specification, we find:

> Numbers will be printed in a field of width *N*. *N* must be between 1 and 26. If the value to be printed out is not an integer, an error message will be printed.

We seem to have found our bug. When we ran the program only two values were printed out:

```
DAY, RATS VS BATS 1                    40
FO ERROR AT LINE 360
```

The integer 1 corresponds to day 1, and the integer 40 corresponds to 40 rats. The program bombed when it tried to print out the number of bats. Suppose, as in the first example, that *B*, the number of bats, was equal to 4.799999, a floating-point number rather than an integer. If such were the case, and in view of the user's manual discussion on integer formats, we would expect to see exactly what we did see.

If we wish to use an integer print format we must convert *R* and *B* to integers:

```
358 LET R=INT(R):LET B=INT(B)
```

Running the program with these changes, and using the same input data gives us:

```
DAY, RATS VS BATS 1          36          5
DAY, RATS VS BATS 2          66          4
DAY, RATS VS BATS 3          42          2
DAY, RATS VS BATS 4         330         12
DAY, RATS VS BATS 5         258         12
DAY, RATS VS BATS 6        2046          7
DAY, RATS VS BATS 7        1803          7
DAY, RATS VS BATS 8       19371         29
DAY, RATS VS BATS 9       19197         29
DAY, RATS VS BATS 10     209253         20
```

The output is considerably easier to read and we are at last dealing with whole rats and whole bats. Suppose we continue with the output:

```
DAY, RATS VS BATS 15          22754953              84
DAY, RATS VS BATS 16          250298940             302
DAY, RATS VS BATS 17          250297130             302
DAY, RATS VS BATS 18          2753248500            302
DAY, RATS VS BATS 19          1651948000            302
DAY, RATS VS BATS 20          18171408000           1812
DAY, RATS VS BATS 21          -2019155600           -302873340
DAY, RATS VS BATS 22          -2221071600           -199896440
DAY, RATS VS BATS 23          -613015800            -199896440
DAY, RATS VS BATS 24          6449990800            -1079440700
DAY, RATS VS BATS 25          -91953450             -1079440700
```

Previously we encountered fractional rats but now we have negative rats. Neither makes much sense. Referring to the program listing we can see that it is the negative rats which cause the bats to also be negative. This occurs at one of our original patches:

```
335 IF B>R/R3 THEN LET B=.9*(R/R3)
```

If the number of rats *R* becomes negative, then *R*/*R*3 will also be negative. The number of bats *B* will start out as a positive number, definitely greater than *R*'s negative value. Thus the test at line 335 will be met and the number of bats will be reset to 0.9*(R/B), or:

```
B =0.9*(R/R3)
  =0.9*(-2019155600/6)
  =0.9*(-336525933.3)
  = -302873340
```

Checking with the output from the program, this is precisely the quantity of bats calculated for day 21. The patch for this can be fairly uncomplicated, the only difficulty being, once again, whether the simplifying assumption that we are being forced to make is going to be valid. Suppose we decide that if *R* ever becomes less than zero it should be reset to its initial value:

```
311 IF R<=0 THEN LET R=S
```

S is a storage variable which saves the value of *R* immediately after it is read in from the keyboard:

```
75 LET S=R
```

If we run the program now we get:

```
DAY, RATS VS BATS 1          28          4
DAY, RATS VS BATS 2          44          4
DAY, RATS VS BATS 3          20          2
DAY, RATS VS BATS 4          88          10
DAY, RATS VS BATS 5          25          3
DAY, RATS VS BATS 6          46          3
DAY, RATS VS BATS 7          28          3
DAY, RATS VS BATS 8          110         18
DAY, RATS VS BATS 9          2           0
DAY, RATS VS BATS 10         22          0
DAY, RATS VS BATS 11         22          0
DAY, RATS VS BATS 12         242         0
```

A relatively minor change to the program has resulted in a major change to its output. The rat population does not increase as dramatically as it did in the earlier example, but that could be due to chance events taking their toll. The bat population was reduced to zero by the ninth day and was unable to recover.

Normally this would be the time we would rerun the program with our debuggers turned on, but unfortunately that is the one thing we cannot do; every time we run the program we will get different results because of the two random number statements. We could try to force execution by writing in some statements that would bypass the random portions of the code, replacing them with exactly specified "chance" values. But there are 10 possible chance paths and 7 debug snapshots per cycle; since in the present example it took 8 cycles for the bug to show itself, the debug effort would be at the least tedious.

In this case our best bet is to play computer. We know that when the program reached line 360 of the eighth cycle the values of *R* and *B* were 110 and 18. The next cycle, cycle 9, would not be a reproduction cycle for either rats or bats (rats reproduce cycles 2,4,6,8,10, etc. and bats reproduce cycles 4,8,12,16, etc.), so the first thing the program would do in the new cycle is calculate a natural disaster severity quotient and determine which group, rats or bats, would be the group to suffer. Next, it would reduce the rat population by the amount lost through predation to the bats:

```
310 LET R=R-(B*R3)
```

We can substitute in our values for *R* and *B* from the previous cycle:

```
R = R-(B*R3)
  = 110-(18*6)
  = 110-108
R = 2
```

We calculate *R* to be equal to 2 at line 310. Since the print statement at line 360 shows the same value for *R*, we can deduce that whatever the severity coefficient (*D*), the rats/bats pointer (*D*1) was not equal to rats. We therefore bypass all the lines of code from 310 to 332, stopping at line 335:

```
335 IF B>R/R3 THEN LET B=.9*(R/R3)
```

The number of bats *B* equals 18, which is definitely greater than 2/6 or 0.33. The test condition is therefore met and we may calculate for ourselves:

```
B = 0.9*(R/R3)
  = 0.9*(2/6)
  = 0.9*.33
B = 0.3000
```

We recall that day 9 is not a reproduction day for bats, so *B* will not be increased by new additions. At best, if there are no chance natural disasters, we will be left with three-tenths of a bat—and even this meager sum would soon be erased because of the patch we put into the program earlier. We know from Chapter 4 that patches very often have serious consequences in places and in ways not originally anticipated, and this is one such case.

The culprit is line 358, the line we put in so that we could use the integer format when displaying our results:

```
358 LET R=INT(R):LET B=INT(B)
```

Clearly, if we take the integer portion of 0.30, we are left with zero. This explains our disappearing bats. What we need now is a patch similar to the one we used for negative rats:

```
352 IF B<=0 THEN LET B=S2
```

As before, *S2* is a variable which stores the initial quantity of bats immediately after it is read in from the keyboard:

```
85 LET S2=B
```

We should now not expect to find any negative values for either rats or bats. Suppose we run the program once more:

```
RUN
TWO GROUP ECOSYSTEM SIMULATION
SPACE BATS VS MOON RATS
BATS EAT RATS AND RATS EAT PLANTS
INITIAL POPULATION OF RATS 100
INITIAL POPULATION OF BATS 10
RAT REPRODUCTION RATE(DAYS) 2
BAT REPRODUCTION RATE(DAYS) 4
NO. RATS/LITTER 10
NO. BATS/LITTER 5
RATS EATEN/BAT 6
PLANTS EATEN/RAT 4
IS THIS A DEBUG RUN? (RETURN FOR NO)
DAY, RATS VS BATS 1          40        5
DAY, RATS VS BATS 2          99        5
DAY, RATS VS BATS 3          69        5
DAY, RATS VS BATS 4         429       30
DAY, RATS VS BATS 5         249       27
DAY, RATS VS BATS 6         957       27
DAY, RATS VS BATS 7         795       18
DAY, RATS VS BATS 8        7557       64
DAY, RATS VS BATS 9        6455       64
DAY, RATS VS BATS 10      66781       64
```

So far things look good. Skipping ahead to day 20 we see:

```
DAY, RATS VS BATS 20     9565476300      6738
DAY, RATS VS BATS 21     5891331900      6738
DAY, RATS VS BATS 22     43534147000     6738
DAY, RATS VS BATS 23    -34519689000       10
DAY, RATS VS BATS 24    -29010886000       10
```

```
DAY, RATS VS BATS 25          90          10
DAY, RATS VS BATS 26         198          10
DAY, RATS VS BATS 27         138           9
DAY, RATS VS BATS 28         924          54
DAY, RATS VS BATS 29         420          54
DAY, RATS VS BATS 30         739          54
```

In spite of our last few patches the rat population has once again taken on a negative value. We note, however, that this time the negative rats did not drive the bat population figure negative also. It is a good bet that any bug this persistent would show up again if we were to rerun the problem, random execution paths or not. Therefore, suppose we turn on the snapshot debuggers, and try again:

```
RUN
—



—
IS THIS A DEBUG RUN?(RETURN FOR NO) YES
```

The screen clears and the first snapshot appears:

```
SNAPSHOT CALLED FROM LINE 150
DAY=1 RAT INDEX=1 BAT INDEX=1
RATS REPRODUCE EVERY 2 DAYS, YIELDING 10 YOUNG
BATS REPRODUCE EVERY 4 DAYS, YIELDING 5 YOUNG
BATS EAT 6 RATS, RATS EAT 4 PLANTS
LAND FOR PLANTS=10000 NO. PLANTS=0
DISASTER SEVERITY=0    SELECTOR CODE=0
X=8    X1=0
NO. OF RATS=100 NO. OF BATS=10
?
```

The snapshot displays all pertinent data and clearly labels all values to avoid ambiguities. There are no plants yet, since the snapshot was taken at line 150 and the number of plants is not computed until line 300. Similarly, the random disaster variables *D*, *D*1, and X1 are followed by zeros.

As the snapshots flash by, seven per cycle, it is a good idea for us to construct a table of values to follow the calculation's progress:

Day	Rats	Bats
1	40	10
2	24	3.6
3	26.4	3.6
4	10	3.888
5	100	3.1104
6	805.2422	3.1104
7	786.57984	2.79936
8	8467.6205	11.757312
9	8397.9766	11.757312
10	91591.86	11.757312
11	91521.316	7.054387
12	1006268.8	42.326322
13	1006014.8	42.326322
14	11063369	42.326322
15	7744180	42.326322
16	85183186	253.95793
17	51108997	253.95793
18	5.621822E8	253.95793
19	5.6218083E8	228.56214
20	6.1839741E9	959.961
21	5.5655715E9	959.961
22	6.1221223E10	959.961

Suddenly, after 157 snapshots, something quite unexpected shows up:

```
SNAPSHOT CALLED FROM LINE 320
DAY=23 RAT INDEX=1 BAT INDEX=3
RATS REPRODUCE EVERY 2 DAYS, YIELDING 10 YOUNG
BATS REPRODUCE EVERY 4 DAYS, YIELDING 5 YOUNG
BATS EAT 6 RATS, RATS EAT 4 PLANTS
LAND FOR PLANTS=7341.7608 NO. PLANTS=-2.0639957E11
DISASTER SEVERITY=0    SELECTOR CODE=1
X=3    X1=4
NO. OF RATS=-4.6439903E10 NO. BATS=959.961
```

Both the number of plants and the number of rats have gone negative. If we now expand our table of values and continue on, in a short while we learn something even more interesting:

Day	Line	Land	Plants	Rats	Bats
23	320	7341.7608	−2.0639E11	−4.6439E10	959.61
23	330	7341.7608	−2.0639E11	−4.6439E10	959.61
23	340	7341.7608	−2.0639E11	−4.6439E10	−6.9659E9
23	350	7341.7608	−2.0639E11	−4.6439E10	10
24	150	7341.7608	−2.0639E11	−4.6439E10	10
24	250	7341.7608	−2.0639E11	−4.6439E10	10
24	310	9004.3093	−1.7030E10	100	10
24	320	9004.3093	−1.7030E10	−4.2169E10	10
24	330	9004.3093	−1.7030E10	−4.2169E10	10
24	340	9004.3093	−1.7030E10	−4.2169E10	−3.7952E10
24	350	9004.3093	−1.7030E10	−4.2169E10	10
25	150	9004.3093	−1.7030E10	−4.2169E10	10
25	250	9004.3093	−1.7030E10	−4.2169E10	10
25	310	10513.982	1.5584E11	100	10

On day 22 there 6.12211×10^{10} rats. On day 23 there were supposedly *minus* 4.6439×10^{10} rats. We can trace the beginning of the trouble to line 300:

```
300 LET P=P+(L*4E5)−(R*P2)
```

The quantity of rats carried over from the previous day is 6.1×10^{10}, which combined with a land area of roughly 7342 (as we can see in our snapshot) caused the value of plants to go negative:

$$\begin{aligned} P &= P+(L*4E5)-(R*P2) \\ &= P+(7342*4E5)-(6.1E10*6) \\ &= P+2.94E9-3.66E11 \\ &= P-3.63E11 \end{aligned}$$

Working backwards from the printed value for P given in the snapshot (−2.06E11), we can deduce that at the time cycle 23 began there were only 1.57*E*11 plants available, not enough to offset the imaginary negative plants calculated by the program.

We were tripped up by another one of our previous patches, at line 315. Negative plants caused negative rats:

```
315 IF R>P/P2 THEN LET R=.9*(P/P2)
```

Variable *R* came into line 315 as a positive value, which caused the condition to be met and *R* to become negative. We recall that we wrote line

315 into the program to guard against overgrazing by a too-high rat population, and we see again that a patch does not operate in a vacuum: a patch may later have consequences quite unforeseen at the time it is installed.

A few lines later, at line 335, the negative value of *R* causes the bats to become negative also, as we saw in our expanded table of snapshots printouts:

```
335 IF B>R/R3 THEN LET B=.9*(R/R3)
```

Fortunately the bats are saved by the zero-or-minus value clamp we installed at line 352:

```
352 IF B<=0 THEN LET B=S2
```

If we look more closely at our expanded table of values we note a few inconsistencies. For example, the table indicates a snapshot taken at line 350; this was true in the original version of the program before we installed the zero-or-negative value clamp at line 352. For clarity's sake we should really change line 355 to read:

```
355 IF Z=1 THEN LET Y=352:GOSUB 1000:GOTO 370
```

The proper line number would then be displayed in the snapshot. A similar situation exists for the snapshot called at line 335.

The rats were not caught because even though we installed a similar zero-or-minus value clamp for *R* we did so at line 311, just two lines *before R* became negative.

It takes two cycles for the program to correct itself, and it does so in an unusual way. First, the large negative value for *R* causes *L*, the land area, to increase from 7341 to 10514; 514 square miles more land than is even possible. In addition to negative animals, the program is creating land mass from thin air:

```
290 LET L=L-(R*3.58E-8)-(B*1.0E-7)
```

This sudden increase in land area, coupled with the massive infusion of plants caused by the noneating of -4.2×10^{10} rats, finally sends the plant population back into the realm of positive numbers at line 300:

```
300 LET P=P+(L*4E5)-(R*P2)
```

The very next line, 311, is our zero-or-minus value clamp on *R*. This time its effect holds since there are no more negative plants to unbalance the system. The rat population is reduced to its initial value of 100, and processing continues. Everything is as it should be save that the moon has permanently expanded by 514 square miles.

The way to prevent this scenario from repeating is to move line 315 to line 322, thus preventing the rat population from becoming so large that it drives the plant numbers negative. Line 315 was a patch which was intended to do just that but we neglected to place it *after* the rat population was increased by reproduction.

Suppose we now run the program again:

```
RUN
TWO GROUP ECOSYSTEM SIMULATION
SPACE BATS VS MOON RATS
BATS EAT RATS AND RATS EAT PLANTS
INITIAL POPULATION OF RATS 100
INITIAL POPULATION OF BATS 10
RAT REPRODUCTION RATE(DAYS) 2
BAT REPRODUCTION RATE(DAYS) 4
NO. RATS/LITTER 10
NO. BATS/LITTER 5
RATS EATEN/BAT 6
PLANTS EATEN/RAT 4
IS THIS A DEBUG RUN?(RETURN FOR NO)
DAY, RATS VS BATS 1          40          5
DAY, RATS VS BATS 2         110          4
DAY, RATS VS BATS 3          86          3
DAY, RATS VS BATS 4         748         12
DAY, RATS VS BATS 5         676          8
DAY, RATS VS BATS 6        6908          5
DAY, RATS VS BATS 7        6878          5
DAY, RATS VS BATS 8       75328         30
DAY, RATS VS BATS 9       75148         30
DAY, RATS VS BATS 10     742183         30
```

The program starts out well. Suppose we skip forward a bit and see if the system is still stable:

DAY, RATS VS BATS 50	361382800	54207419
DAY, RATS VS BATS 51	36138290	3252446
DAY, RATS VS BATS 52	182859750	19514676
DAY, RATS VS BATS 53	65771690	9865753
DAY, RATS VS BATS 54	72848892	8879177
DAY, RATS VS BATS 55	19073830	2861074
DAY, RATS VS BATS 56	20981246	15449800
DAY, RATS VS BATS 57	100	13
DAY, RATS VS BATS 58	242	13
DAY, RATS VS BATS 59	164	7
DAY, RATS VS BATS 60	1342	42

The program looks as though it can take care of itself now; indeed, if we were to follow the output for a much longer time we would see that at no time do zero or negative values appear.

As for the validity of the data, that responsibility lies with the person who formulated the model. We have seen that, especially in the present case, a computer model of natural events involves assumption, simplification, and compromise.

We have also seen how the many elements of the programmer's craft must work together in a successful debugging effort. No single technique is perfect for all applications and no method has any greater claim to fame than any other method. But used in concert with one another, and under the guidance of a logical plan of action, a few simple techniques are all that are required to debug even the most complicated of programs.

6

Hardware Bugs

On occasion you may encounter a strange bug that defies every debugging effort you try, leaving you with the suspicion that something is wrong with the equipment or hardware. To clear up such doubt, you may be wise to run a test program or one that has operated successfully in the past. If the test runs smoothly, the hardware is pretty much vindicated. But, if the test program fails, you probably have a bug in the hardware or peripherals. Locating the cause of a malfunction in the hardware can be tricky—even if you're a technician! In any case, an understanding of how peripherals work will help you track down the cause of the problem.

TELETYPE

The workhorse of the computer peripheral industry, and on that is extremely popular with hobbyist-type computers, is the famous teletype. These tele "printers" contain their own keyboard so that you have a complete computer terminal with capabilities of tramsmitting and receiving. The most popular and easy to interface to a microprocessor

system is the Model 33 ASR Teletype. The ASR is a paper tape punch and reader. However, for personal computer systems which use magnetic tape storage, an ASR unit would not be necessary. Because of the extreme popularity of the teletype and the fact that is an industry-wide standard, programs are offered in teletype-compatible punched paper tape. Often the company making software available will use one kind of magnetic tape system, such as the Kansas City Standard, and then offer punched paper tape as an alternative. High-speed tape readers, which utilize semiconductor light sensitive pickups to detect the presence of the holes in the punched paper tape, have been offered for microcomputer hobbyists since the days of the Altair 8800.

It is possible to purchase a Model 33 teletype without the keyboard coding assembly. The printer only teletype is known as an "RO33" or Read Only 33. And when they can be found, they will generally sell for under $400. The printer mechanism for the Model 33 prints on 8½"-wide paper and is capable of printing a total of 72 characters (in a printer this is called a "72-column line"). Because of the electromechanical setup in a printer using a Model 33, however, it is not possible to mechanically backspace. Thus the teletype does not make a good typewriter since you cannot backspace and correct the printed page. When a paper tape punch reader is used, however, it is possible to use the rub-out symbol in conjunction with an actual backspace button on the tape, and thus produce an error-free punched paper tape which can be put on the reader of the teletype to print an error-free page.

While the ruggedness and reliability of the Model 33 teletype has been proven over many years of use, there are some major disadvantages. The primary disadvantage is that the baud rate is fixed at 110 baud. At 110 baud or 10 cps (characters per second), the printout rate is fairly slow. There is no way to modify the electromechanical mechanism to allow the machine to operate at any different speed. This is to be contrasted with the later model teletypes which are available now that will operate at various rates and which use a dot matrix printhead with over-strike capability. These sophisticated machines are quite expensive however, and with a minimum of accessories a good price for a new one would be well over $3,000.

The ancient teletypes used five-level punched tapes (rather than seven-level ASCII code) and ran at only 60 words per minute. It is the five-level code that causes extreme problems. It is called a "Baudot code," and because it does not have an adequate number of bits to be able to represent the entire ASCII character set, Baudot machines use shift commands prior to the reception of the letters and figures. When the receiving teletype picks up one of these LTRS commands (letters), it mechanically shifts the teletype to make letters. The same holds for figures and punctuation characters. When the teletype receives the FIGS command, it shifts into position electromechanically so that all the

characters that follow will be numbers. There is, of course, absolutely nothing wrong with the Model 15 teletype. It's just that interfacing it with a microcomputer system is rather difficult.

The teletype character recognized by the standard Model 33 is a serial data character which consists of seven bits plus parity plus a start and two stop bits. It is transmitted at 110 baud which is just under 10 milliseconds. If you're wondering why the 110-baud figure, quickly add up seven bits plus parity equals eight bits, then three bits making up start and stop make eleven bits total to represent one word. To achieve the 10 cps rate it would be necessary to transmit the character (baud) at 10 times the bit rate. Ten times eleven bits equals 110 baud. Out of the seven bits of data that are transmitted, the least significant bit goes out on the serial data line first. In the standard 20-milliamp current make/break interface that is supplied with the Model 33 teletype, a mark or 1 is a current on condition and a space or off condition corresponding to logic 0 is an open condition. The reason it is dubbed a "20-milliamp loop" is that you are detecting whether or not the 20 milliamps of current is flowing through the circuit. The teletypes can be interconnected so that several can be hooked in series on one loop and fed by a common 20-milliamp power source, and any of the teletypes on the loop will interrupt the loop and transmit to the other machines on the loop. Some automatic identification information built into the teletype keyboard makes use of the "Here Is" key, so that information on the source teletype unit can be transmitted to all the teletypes connected on the loop. In standard computer interface configuration where the teletype is serving as computer I/O, chances are there would not be more than one teletype connected on the loop. The loop in turn would connect to the computer IC circuits. Since the teletype itself does not contain its own 20-milliamp power supply, the 20 milliamps will have to be supplied by the computer I/O circuits.

A widely used interface standard which can be built into the Model 33 teletype and which has become a standard for serial data interconnection on personal computer systems is the RS-232C standard. Rather than using currents, as with the 20-milliamp loop, system RS-232C uses voltage levels with 0 defined as +3 to +9 volts or more and 1 as -3 to -9 volts or less. Now all the peripherals that are interconnected on the same RS-232C serial data line will be connected in parallel. All the peripherals on the RS-232C line may be connected in parallel with the transmitting unit being the one that will supply the logic 1 voltage pulses to the computer or other peripheral.

CRT TERMINALS

CRT terminals that interact with computers are limited usually to character displays which can be alphanumeric and special characters,

as well as foreign characters. There are also graphics systems available which produce line drawings of objects; some of these are just now becoming practical for the personal computer system.

Figure 6-1 shows a block diagram of a CRT terminal. Note that the terminal itself is shown with I/O circuits that allow it to interface with any computer which has the capability of interfacing with any other serial terminal, such as a teletype (TTY). Thus the CRT terminal appears to the computer to be a teletype. There are boards which plug into the personal computer system which provide the video output to a simple CRT, but these are usually not called "terminals" because of the dependence on the internal workings of the host computer. Commonly, these are simply called "CRT driver boards." Their advantage is that data is written on the screen almost instantaneously. They invariably use some sort of Direct Memory Access (DMA) system. This means that the display memory is written and read (accessed) directly by the MPU as if it were part of the computer's memory. In some extremely low cost CRT driver interfaces, the computer's memory also serves as the display memory. These systems usually allow you to do such things as generate a multiple cursor. Multiple cursors can provide coarse graphics display. To display a line, you merely generate a number of cursors side by side.

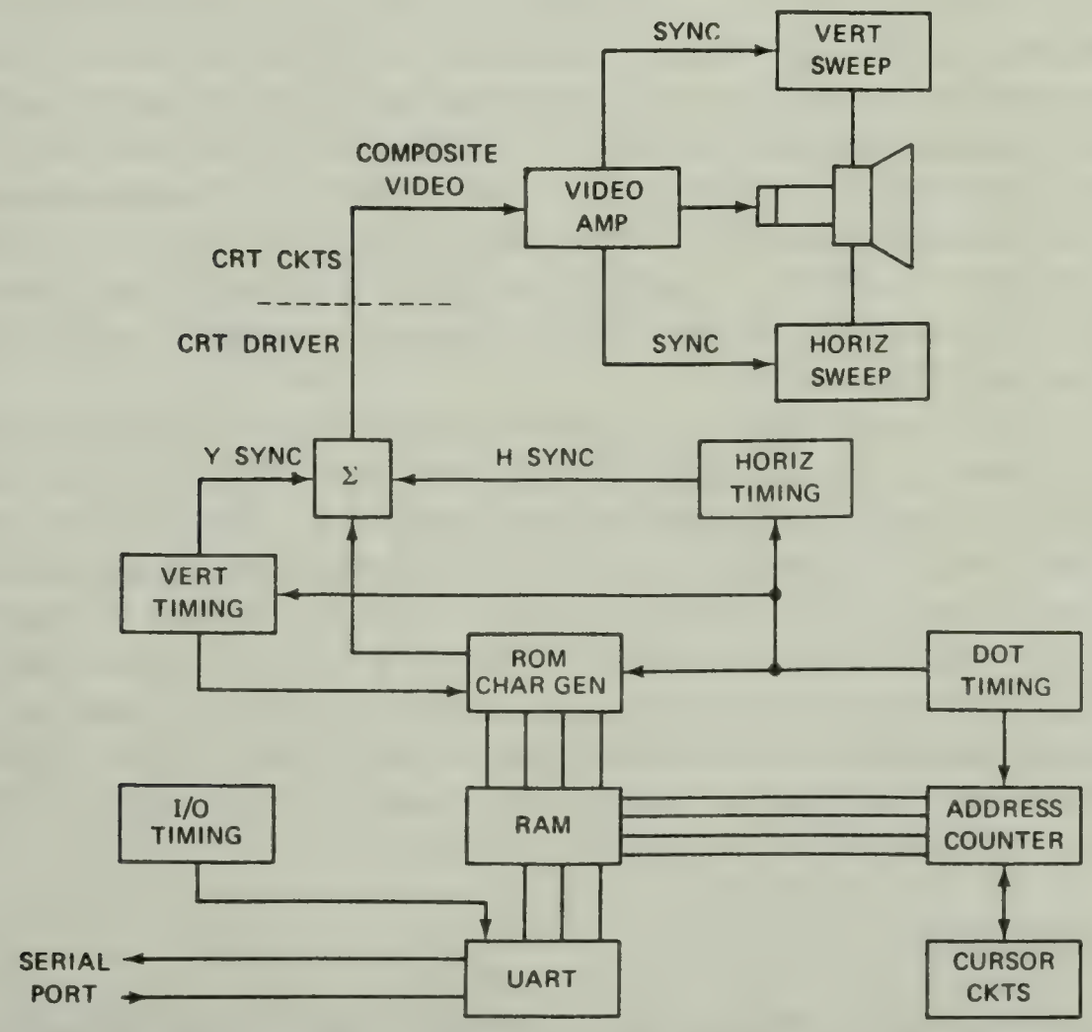


FIG. 6-1. CRT-terminal block diagram.

Circles are actually a series of cursor blocks with the edges touching to arrange a circle. Close inspection of the circle makes the block-type line segments clear but from a distance the circle looks as if it were drawn with a fat pen.

True graphics in a CRT display are given by using vectors to draw straight lines of various lengths on the CRT. By shortening the line, it is theoretically possible to light only the area of the screen created by the size of the electron beam and phosphor dots. This area illuminated only by the width of the modulating electron beam is called a "pixel." A number of pixels are arranged like the block cursors so that they create either a line or a character (as shown in Fig. 6-2). Lines of light are swept across the screen of the CRT continuously, but the intensity of the electron beam is kept at such a level that the lines do not cause the phosphor screen to light up. When the timing circuits in the display driver circuits determine that a spot is to be illuminated, the video signal goes positive. Since the CRT is actually like any other electron tube, and the video signal is applied to the grid of the tube, it conducts more when the video signal goes positive. This action lights up the screen creating

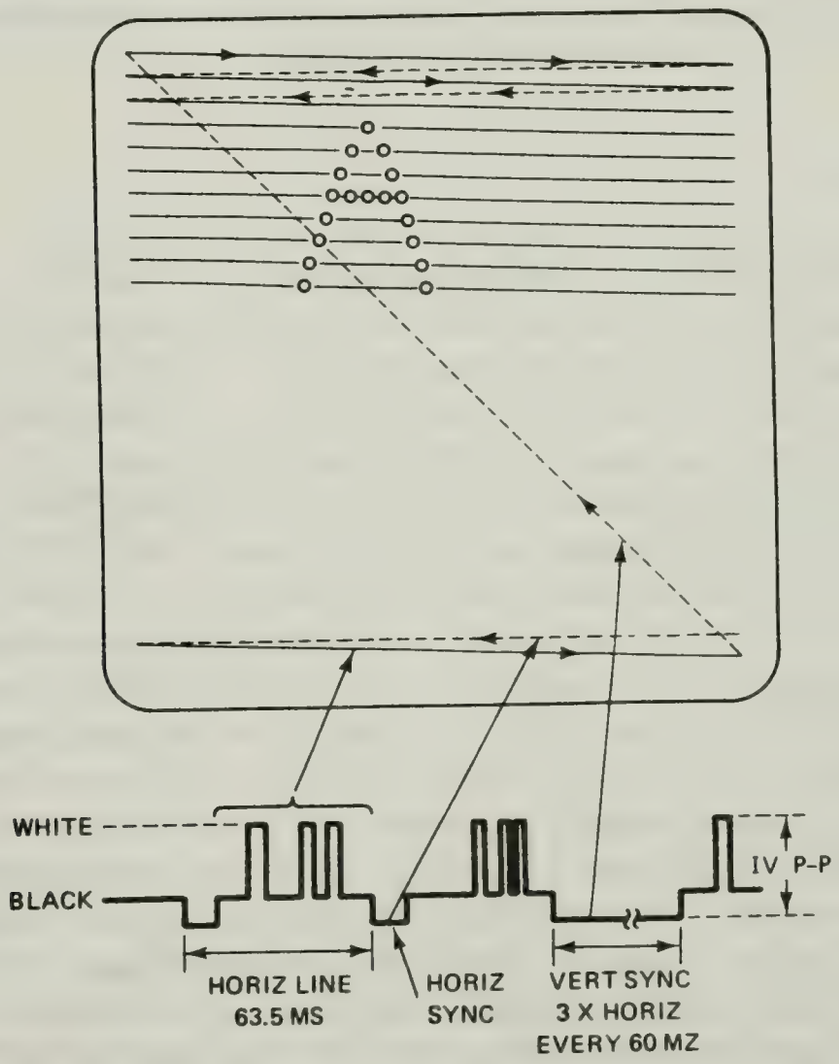


FIG. 6-2. CRT screen, showing the scanning action of the electron beam and the composite video signals in relationship to the overall picture.

the dots that will form the lines making up the character. As the electron beam sweeps back at a much higher speed it is cut off (retrace blanking). The next electron beam sweep creates the next row of dots making up more of the character. Again the retrace is blanked. Blanking occurs at the end of the horizontal line and is called the "Horizontal Sync Period." The CRT driver circuits detect the horizontal sync signal which is a low or even negative voltage, and this sync pulse is used to keep the sweep in sync with the timing circuits. When the sync/blanking signal combines with the video signal and the vertical sync pulse (formed by counting the proper number of horizontal lines), the resultant signal is called the "Composite Video Signal."

Some personal computer systems further use the composite video signal to modulate an rf (radio frequency) carrier signal that is tuned to some unused TV channel. In this way, it is possible to display the output of the personal computer system on a standard home TV set without making any internal modifications to the set itself. CRT displays are relatively inexpensive ways to display a lot of data. A CRT terminal doesn't give hard copy printouts of programs, however, and once the data is off the screen, it is lost forever. Fortunately, there were already some low cost printers being made for minicomputer users; unfortunately, they still cost several hundred dollars.

LINE PRINTERS

Southwest Technical Products Corp. was the first to introduce a limited ability, limited price printer for the user of the microprocessor as a personal computer. It has some limitations; for one, it only prints up to forty characters on a line (forty columns). It uses a dot matrix print head, however, and print quality for the device is pretty good.

The dot matrix printer principle is fairly simple. It works somewhat like the CRT display except that rather than creating the character a line at a time, the characters are created a column at a time. Seven (or more) tiny rods are connected to solenoids which are driven as shown in Fig. 6-3 by timing circuitry similar to that in the CRT display. As the timing circuits dictate, rows of dots are formed at the same time and all proper solenoids are driven for creating the given character in the given column. After five columns are formed, the character is completely shaped and the printer spaces two column times to give the proper space between characters. After printing to the end of the line, the paper is advanced and is moved to the next character starting point plus two "row" spaces to properly space the distance between lines. The left margin (called the "home position" for the print head) is detected by operating a mechanical or optical switch that signals the printer control circuits that a line has been printed and initiates the paper advance.

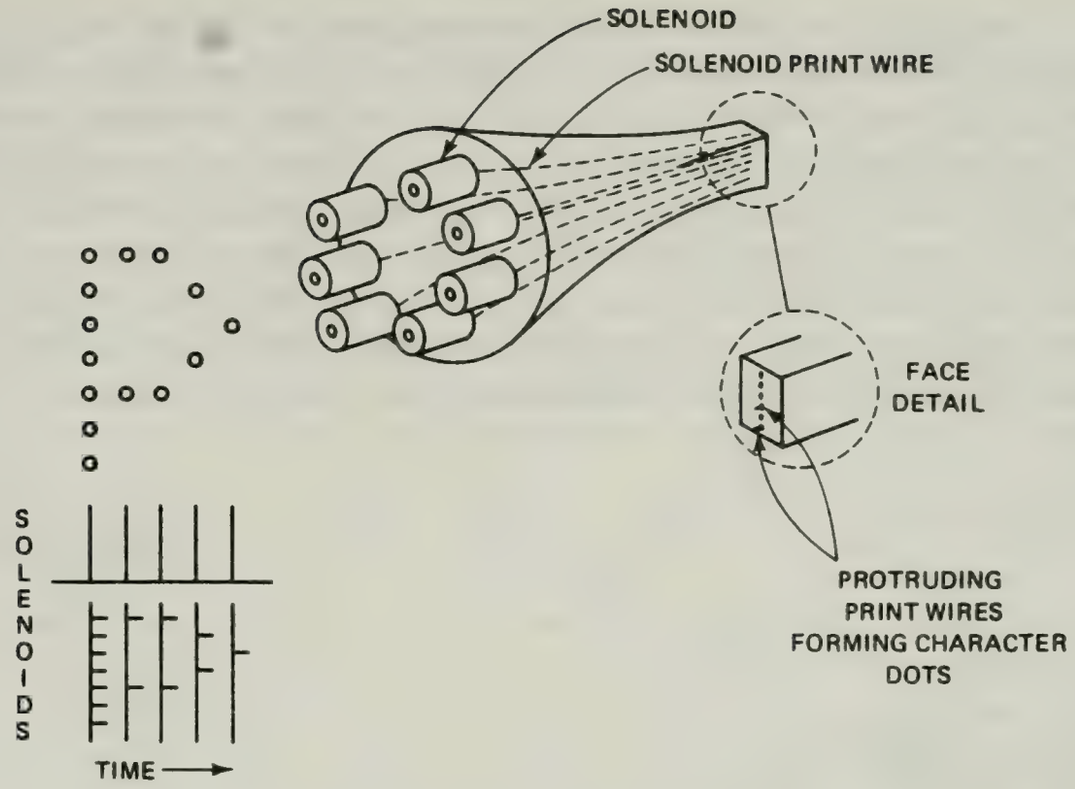
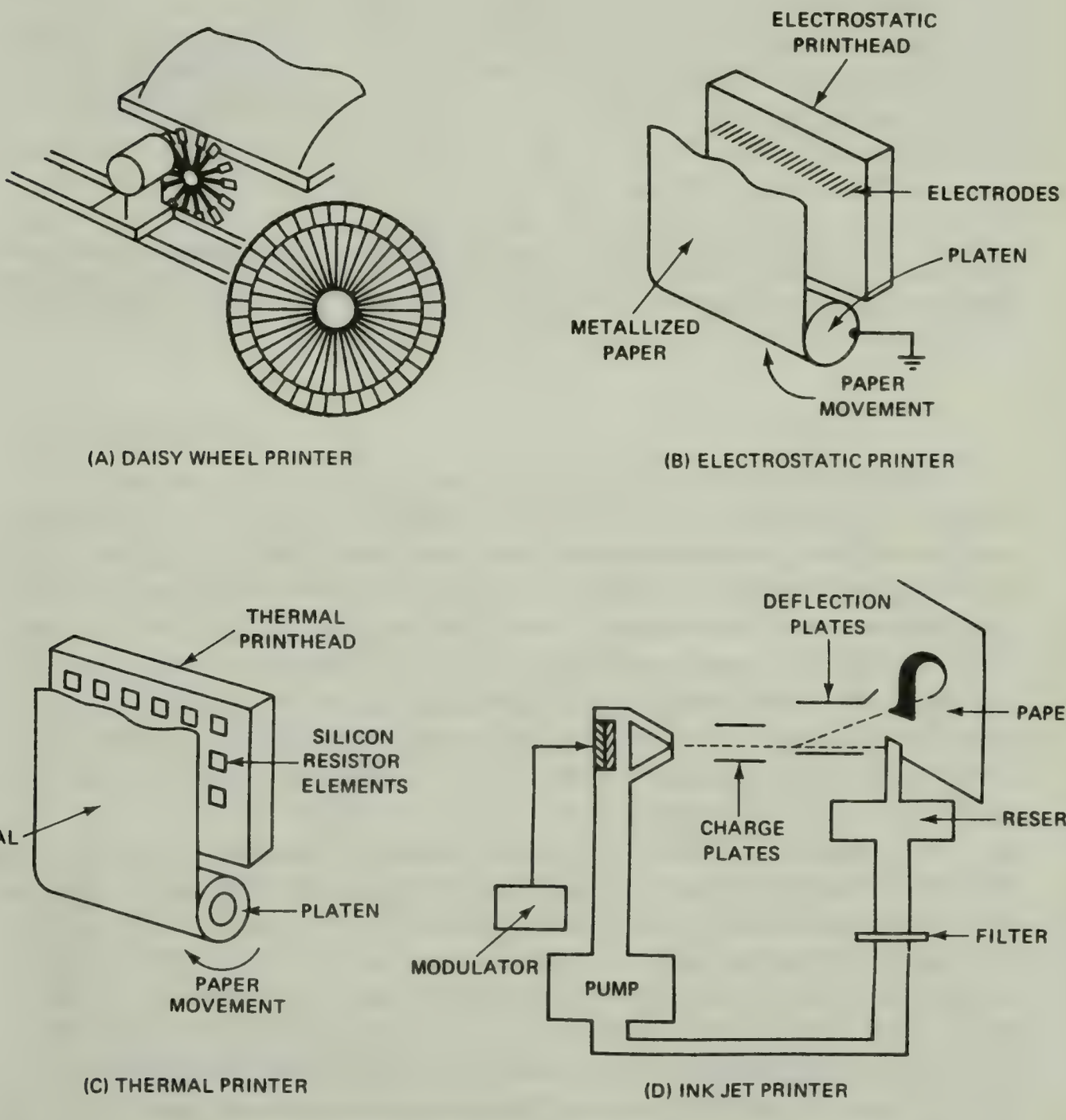


FIG. 6-3. Drawing of the dot-matrix printhead and timing diagram showing the formation of the letter "P." Similar to the solenoids, the needle printer uses tiny needles, but instead of the solenoids, electrically operated relay-like hammers bang the needles into the ribbon and form the dot-matrix character on the paper.

Because you must print a whole line of data in this type printer, it is called a "line printer." There can be no character at a time printing like you get with a typewriter because the head has to keep moving once it is started. The line can be filled on the left or right with spaces to inhibit printing except in some desired area, but you do not see the characters until a print command sends the print head across the paper in one complete line. Some line printers are capable of striking over a line that has already been printed. Other schemes exist for allowing a shift of the head or the platen (paper roller) to allow use of two-color ribbons for such uses as in cash registers.

The problem of hard copy for a personal computer system is an age-old problem. We have seen how a dot matrix printer setup works—there are some other types of printers (shown in Fig. 6-4) which should be mentioned at this point. There are basically two types of printers: impact and nonimpact. Impact printers include cylinder, ball, wheel, dot matrix, daisy wheel, and chain or band type. There are also drum types, although they are used generally in smaller calculator-type units. The nonimpact printers include thermo printers, which cause printing on special thermal sensitive paper; electrostatic printers, which use special metalized paper; and finally, ink jet printers, which spray a jet of ink using a high voltage deflection system similar to that in a CRT.

The cylinder or ball or wheel printing devices are among the most common impact printing mechanisms. These are the printers that are used in the Model 33 teletype, for instance. In these devices a character-studded rotating ball, cylinder, or print wheel is controlled electromechanically and is either hit by a hammer or directly strikes the ink ribbon or carbon film ribbon creating the impression on the printout paper. There are several advantages to this type of printing: multiple copies are easily produced, the printing is high quality depending on the characters that have been imbedded into the wheel or cylinder, and the



(A) DAISY WHEEL PRINTER

(B) ELECTROSTATIC PRINTER

(C) THERMAL PRINTER

(D) INK JET PRINTER

FIG. 6-4. A group of different printing mechanisms that may be used now or in the future with personal computer systems. All but (A) are non-impact printers, and the function of each, along with a comparison of systems, is covered in the text.

cost of this type printing arrangement is relatively inexpensive. These printers use standard paper. Most of the units produce eighty or more columns, and they have easily changed character fonts. On the other hand, because they are electromechanical devices, these printers are usually quite noisy and are extremely slow. They're also mechanically complex and although durable and reliable, when servicing becomes a problem one must generally have some specialized training in how to repair these devices. It is impossible, too, by using this type system to be able to produce graphics on the paper should there ever be a need. And finally, because of the electromechanical setup, carriage control is generally not available. This means, as mentioned in the discussion of the teletype, it's impossible to backspace and print over characters with the TTY-type impact printing mechanism.

Daisy wheel printers mix the best of both worlds, that of a solenoid of the dot matrix printer and the solid character of the impact-type printer. Here the solid characters are contained on a print wheel which has ninety-six spokes. Each character is embossed on a tip of the printwheel. The print hammer or solenoid strikes the desired tip to produce the printed letter. The print head itself looks somewhat like a flower petal or daisy wheel, and electromechanical drivers spin the correct character into firing position for the print hammer to hit. The daisy wheels are easily interchangeable so that fonts can be changed very easily; thus the daisy wheel printer finds itself in word processing applications where art work is to be prepared by the printer itself and where various fonts may be chosen to put together a printed page. For instance, using a daisy wheel printer, along with some sort of editing CRT, one is able to produce an error-free page. By typing in a computer, one can also justify the type on the right hand of the page and subdivide the page into two or more columns. A sheet of specially treated white paper is inserted into the daisy wheel printer carriage mechanism and an absolutely correct page is printed out. On this page, however, where there are to be italicized characters, white spots are left. At the bottom of the page when the camera-ready characters have been printed, the daisy wheel is changed to the italic wheel and again under word processor control, the printer prints short lines of italicized characters which are designed by the word processing machine to precisely fit into the blank spaces left in the printed page. The person preparing the art work cuts out the italicized words and glues them in place on the printed page. You now have a camera-ready piece of art work which is ready to be printed into a format such as this book. A variation of the daisy wheel has been recently introduced by a company which calls their printer a "Spin Writer." It uses the same daisy wheel petal-type arrangement except that the spokes are bent upward so that the print wheel itself is actually a cup shape. This cup-shaped wheel spins in front of the print

hammer solenoid the same as in the daisy wheel printer. The spin writer offers the same advantages as the daisy wheel. Of course, the company who sells it claims it has benefits over the daisy wheel printer. At any rate, the advantages for a daisy wheel or a spin wheel-type arrangement are that multiple copies are easily produced and that you can use standard paper. Also, it operates at an incredibly high speed, it is relatively quiet and has excellent print quality. However, its cost ranges from moderate to high, and it requires sophisticated drive electronics to line up the daisy petals. Like other impact printers that use discrete characters, these do not have the graphics capabilities of, say, the dot matrix-type printer.

Most giant computer systems contain one or more chain or band printers because of their exceptionally high print speed. A closed chain or metal band containing multiple sets of characters spins on drive wheels at an oval path behind the printing paper. In front of the paper sits a bank of impact hammers. The printer drives each hammer at precisely the instant that each desired character appears in its impact window. Printing appears on either the front or the back side of the paper depending on the ink source. This type printer has exceptionally high speed, over 1200 lpm (lines per minute), uses standard computer paper, can produce at least 135 columns, and has multiple copy capabilities. However, all the components in the chain or band printer are extremely high cost and mechanically sensitive. Character fonts obviously are difficult to change. The print quality is only fair, graphics are out, and the printers are moderately noisy. However, they are so fast that in a big computer center they are almost a necessity.

Of the nonimpact printers there are the spark gap or electrostatic printers. These use a low cost stationary print head but require extensive drive and interface circuitry. However, they are extremely low cost which makes them very appealing to the hobbyist market.

Metalized paper is moved past a row of electrodes. At the precise moment of contact, the entire bank of electrodes fires producing a row of spaced dots. The paper advances one step and another row of dots is printed and this process continues until an entire block of characters has been created. By mounting several print heads side by side it is possible to create a forty-or-eighty-column printer capable of writing an entire line at a time. The advantages are obvious: extremely high speed (some of these units approach chain printers in performance), also low cost. Because of the versatility of using the tiny pin-like electrodes, these printers easily produce virtually any size or style of character and are very quiet in operation. The only mechanism that is moving is the metalized paper. Since the size and style of character are easy to reproduce, so are graphics. Mechanical failures are reduced because of a minimum of moving parts. The disadvantages, on the other hand, aren't nearly as important as the advantages. The major disadvantage is that

since it requires special paper, sometimes the print quality can be extremely poor. The units produce a burnt paper smell which is sometimes offensive. The electronics required to drive this type printer are the most sophisticated of all the others. While the unit is unable to make multiple copies, it is able to print fast enough to make several copies in the time in which many printers make one pass at a multiple copy. Another disadvantage is that most of the electrostatic printers that are available on the market at this time use an extremely narrow forty-column width. Nevertheless, electrostatic technology is a sufficient point of change so that electrostatic printers promise to become formidable contenders in personal computer systems' hard copy production.

Another of the nonimpact printers which is extremely popular in small table-top pocket calculators is the thermal printer. These work basically the same way as spark gap printers; however, silicon thermal resistor elements form the segments of the letters. They literally burn characters onto a specially treated paper and because of the extremely low-burn temperature (usually no greater than 100° Centigrade), they can operate fairly fast. The heating elements can be configured as dot matrix or dot row, and these printers because of their quiet operation enjoy varying degrees of popularity. Their cost is low to moderate; the fonts are easily controlled; operation is very quiet. Proper heat sinking of the thermo print head can achieve moderate speeds with high mechanical reliability and fair-to-good print quality. Depending on the print head, they are capable of producing clean graphics. However, the most common thermal print heads are less than forty characters wide, and again they use a special paper. They cannot produce multiple copies. The silicon resistor element sometimes wears unevenly creating uneven characters, and the special paper will not retain character readability in high sunlight. The special paper can also lose characters when exposed to high heat and humidity.

The final nonimpact printer that has been mentioned is the ink jet printer. At least one company has come out with a fairly low cost ink jet printing mechanism. This works on the same principle as in the Millikan oil drop experiment. A charged glob of ink forced from the reservoir flies through the ink injector and then is deflected. By carefully controlling the angle of deflection, the glob of ink can be make to hit the paper in the exact desired position. By using a pulsed rapid injection system and controlled deflection angle, the ink jet printer creates a dot matrix-like character. The entire ink jet mechanism rides along the carriage to produce the desired length.

Basically, an electronic transducer causes a mechanical reaction to squirt an ink drop from a nozzle. The ink drop passes through a charged electrode which causes the ink drop to take on an electrical charge and then past high voltage deflection plates which either repel or attract the

charged ink drop. The drops that are not deflected to the paper are caught and then reused through a special reservoir. The advantages of ink jet printers are exceptionally high speed (over 1,200 lpm on some models), the ability to use a wide variety of papers, and extremely quiet operation. Most units produce eighty or more columns, printing quality is good, and graphic fonts are easily produced. The disadvantages at the present time include high cost (although it appears the technology is going to allow a cost reduction). These units, like other nonimpact printers, cannot produce multiple copies. There are certain mechanical reliability problems that come into play because of the complex deflection plate mechanism, too. Also, the injector sometimes clogs but technology will probably come to the rescue there, too.

More than one manufacturer sells a needle printer for under $600. The needle printer works on the same principle as the dot matrix printer except that the dots are formed by banging on the back of some little needles to form the characters. The low cost printers now available give the personal computer user some useful peripherals that help cultivate a further exchange of software. Lower cost printers are in the works now and promise to do even more for personal computer systems.

MAGNETIC TAPE MEMORY

One of the earlier hobbyist computer magazine publishers, *Byte* magazine, sponsored a get-together for personal computer system manufacturers. The group met in Kansas City to try to arrive at a standard for the storage and retrieval of programs for hobbyists. That group came up with a method for storing digital data on standard Phillips cassette tapes by using an ecxternal circuit with a standard, moderately priced cassette recorder. The Kansas City Standard is still in wide use today. It allows recording digital data on machines with widely varying tape speeds. This means there can be an interchange of programs via cassette tapes recorded to the Kansas City Standard. Suppliers went right to work providing the interface circuits between the computer and the tape recorder. With the cassette mass storage device you could write a program for your computer, save it on the cassette, and anytime thereafter, run the program by loading from the cassette into the computer.

Although the Kansas City Standard is widely used among hobbyists, there are other types of magnetic tape computer storage methods available. The greatest majority of these use standard Phillips-type cassette tapes for data storage. The differences come in how the magnetic fluctuations (for creating logic 1 and logic 0) are stored on the magnetic tape. Special digital recording heads allow the recording of logic 1's and logic 0's directly as digital data. These heads are built for

low distortion so that each transition to a logic 1 creates a positive output from the magnetic head. Driver circuits for the magnetic head are interconnected in such a way that a logic 0 is recorded in the opposite direction, thus magnetizing the oxide on the tape in the reverse direction for logic 0. When the tape itself has passed by a pickup head or "read" head, the output pulses are either positive or negative depending on whether a logic 1 or logic 0 was recorded on the tape. This type of recording system is called "phase modulation" and is even used in floppy disc systems. One of the simplest coding systems is called "NRZ1" (Non-Return to Zero, change at logic 1). The recording track is divided into small segments all the same length. For instance, if the recording is made at a bit packing density of 100 BPI (bits per inch), each segment would be 1/100 of an inch or .01 inches long. The read electronics are gated so they only read signals which come shortly before to shortly after the dividing line between segments. In other words we're not interested in the duration of the magnetic field itself, only in the short time periods between each recorded segment. As shown in Fig. 6-5, during this transition period if there is a magnetic transition from one direction to another, a pulse will appear in the gate. The presence of a pulse indicates a logic 1 and the absence of a pulse, a logic 0. These recording systems are fine for a fancy digital head data system but what about using an inexpensive audio recorder?

Audio recorders without special heads are very fussy about what they are willing to accept as input signals. They are designed to exactly reproduce whatever signal caused the original recording. Therefore, the audio people attempt to compensate for any nonlinear response in the tape head both on record and playback to make response more or less flat over the entire audio spectrum. While this kind of compensation does affect sine waves, it spreads the harmonic energy in a square wave

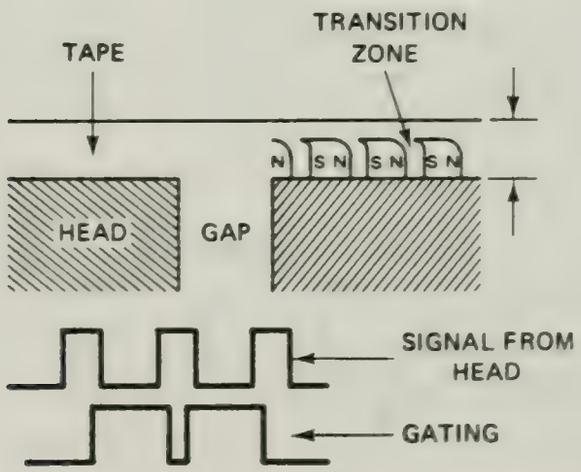


FIG. 6-5. NRZ1 (non return to Zero, change at 1) recording system. Here you see how the magnetic tape is recorded on and how the timing signal is lined up to occur slightly before the end and slightly before the recorded signal. Any transition from a one fo zero will cause an output pulse.

and generally makes a messy wave form. Figure 6-6 shows what happens if a square wave form is recorded on an audio cassette head. Note the impulse on both the leading and trailing edge of the square wave. Play this back, you get a double impulse, one pair at the leading edge and one at the trailing edge of the original square wave. There is no way to get a square wave back for a square wave in. Another problem is that audio recording systems are recorded with bias and with the tape in less than saturation, or in other words, with fairly low signals. Professional digital recording uses saturation recording methods as we have already mentioned. To overcome all these problems, the low cost digital hobbyist system uses a biased FM carrier to avoid the distortion problems inherent in recording with a low cost audio tape machine. Another problem solved by the Kansas City Standard was to compensate for differing speeds between low cost audio cassette recorders.

Using the Kansas City Standard you simply put a signal on the tape that not only tells you where the 1's and 0's are but also how fast the tape is going. The recovered speed information controls the receive timing circuits. The result is called a "self-clocking" recording method. The Kansas City Standard was also designed especially to operate with a low cost, easily available circuit which changed the serial data (bit follows bit) to parallel data similar to that used in the bus structure of microprocessors. This device was called "UART" for Universal Asynchronous Receiver Transmitter. The UART demanded clock at sixteen times the data rate. This means that it has the opportunity to sample each receive bit sixteen times and allows it to compensate for the line hits and other interference which may disturb the normally clean wave form one might expect to receive.

Using the Kansas City Standard to record a 1, you record sixteen half sine waves of a frequency that is eight times the data rate. For a 0 you record eight half sine waves of a frequency that is four times the data rate. Each half sine wave is switched just before its zero crossing is phased, so that the wave form appears to be continuous as shown in Fig. 6-7. If we switched the wave forms anywhere, we might switch

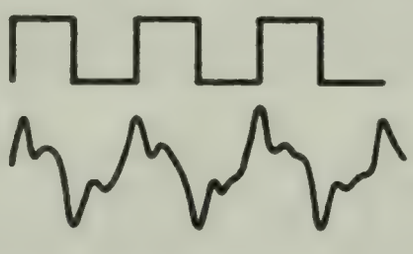


FIG. 6-6. Square wave applied to audio record head. If one tries to record a square wave with an audio head, the problems mount up. You actually get a spike when the square wave goes high and then another when the square wave goes low. Add to this resonance in the circuit, and you get ringing at some frequency.

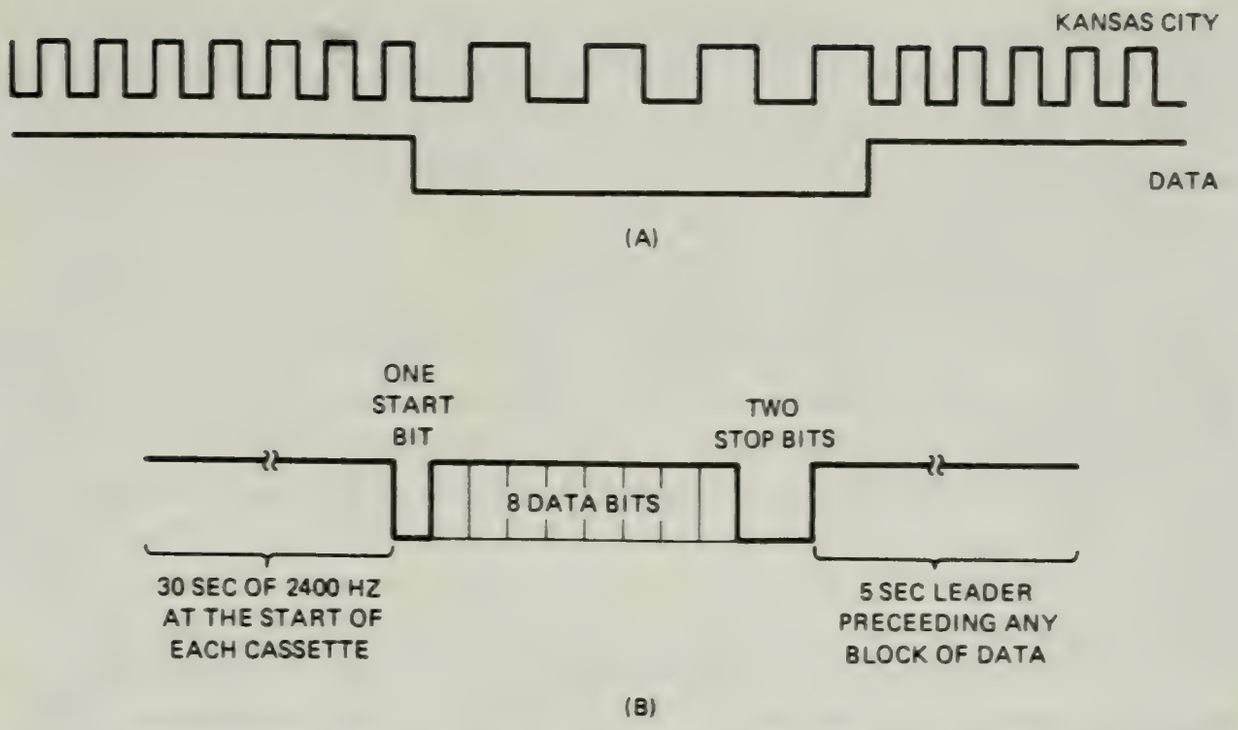


FIG. 6-7. Kansas City recording standard. Part (A) of the figure shows the actual encoding scheme, while Part (B) gives the leader, start/stop standards, etc. This encoding scheme is very popular with computer hobbyists because it works with inexpensive cassette recorders.

one of the analog signals while it was at a high voltage level. If the preceding bit had been at a lower voltage level, it would cause a very fast low-to-high transition similar to a square wave. That would create a pulsed magnetic output on the head as discussed previously. At last, the analog output-signals from the interface are reduced in amplitude to make them compatible with the auxiliary input of a low cost standard cassette recorder.

During playback, signals from the earphone plug on the tape recorder are amplitude limited to minimize tape variations and interference from bias, hum, and other noise. One clock pulse is reconstructed from each zero crossing. The distance between zero crossings is measured and if the distance is too great to be a 1, a 0 output is provided on the data line and a new clock pulse is thrown in for the UART receiver circuitry. Otherwise a logic 1 is provided on the data line and no extra clock pulses are needed.

To compensate for overspeeding which would be caused by recording on a slow tape machine and playing back on a fast one, the characters are output slightly slower holding the character rate down to say 75 or 80 percent of capacity. If overspeeding is present, the maximum rate that would be encountered is still under what a system can accept.

From this, you can see that the Kansas City Standard is extremely tolerant of tape speed differences because it is self-clocking and will tolerate something like a 20 percent variation from normal. This makes

it excellent for use in interchanging programs between personal computer users. It must be kept in mind however that we have covered the recording problem only on a bit-by-bit basis, and that these bits must be organized properly in blocks on the tape in order to be used by one system or another.

We will quickly look at an example of one of the most popular hobbyist tape recording formats. It is a format generated by software located in the MIKBUG debugging program written for Motorola 6800 microprocessor. The data record for MIKBUG is divided into frames. The start of each record is denoted by the recording of an S. Next a 1, 0, or 9 indicates the type of record, 0 being a head of record, 1 being a data record, 9 being an end-of-file record. The data record is the one we are concerned with, so the first two recorded characters on the tape will be S1. The S1 is followed by the byte count which tells how many bytes of data follow. After the byte count comes the address where the data bits are to be stored. For instance if the four following bytes were 0000, it would mean that we will start storing the data at location 0000 and will continue until the data record ends, or until we receive an end-of-file record. The end of the data stream is followed by a checksum number. The checksum number is derived by summing the data bits in another register as they are loaded. When the last data bit has been counted, the checksum is then pulled from the other register and stuck on the end of the tape. Whenever a checksum error is detected, it instructs the MIKBUG firmware to cause the data terminal to print a question mark and stop the cassette player or tape reader.

Obviously, before tapes can be interchanged from one system to another, the user will have to make sure that his tape data formats are compatible with the data formats on the tape he is trying to read into his system. That is why the use of a standard such as MIKBUG is handy, although there are other standards. In many, data is stored in fixed 128 character blocks.

Some years ago, IBM developed a magnetic bulk storage device they called a "diskette." Big magnetic discs spinning at thousands of revolutions per minute had always been popular with the bigs in big computer centers. These "hard discs" as they were known could store millions of bytes of data (megabytes). IBM's so-called "floppy"diskette (because it was flexible, not hard) was used to replace punch card machinery in computer centers.

At first, single-sided, single-drive IBM format (270,000 byte) disc systems cost better than $3,000. Now you can get a dual drive system with dual heads and double density for less than that. The minifloppy, with storage on the order of just under one-half the standard floppy, is available too, but compromises in its performance at this stage make it best suited for applications where space is at a premium.

A typical floppy disc is shown in Figure 6-8 with accompanying details in the following figures. Basically, six essential signals allow communication with the disc drive. These are done generally by electronics controlling the drive called the "disc controller." For recording, there are two major formats: the IBM or soft-sectored format, where the sectors of the disc are defined as "a function of timing in software," and the hard-sectored disc where built-in timing marks control the data storage areas on the disc. In either system, there will be at least one index hole to provide synchronization by means of an optical pickup arrangement. There will be an output from the light pickup for each revolution of the disc for use in synchronizing the soft-sectored format. There may be sixteen or thirty-two output pulses in a hard-sectored format.

Bit timing in a typical disc system is taken care of on the same track where the data is recorded by using a unique recording method, "FM" or frequency modulation, that allows the timing data to be separated as information is read from the disc. If one of the bit cells contains only one switching transition as shown in the figure, then the bit is a logic 0. If there is the customary one bit of timing followed by another bit (data) the data stored on the disc is a logic 1. In this way the logic 1's and 0's representing data are taken from the disc. At the same time the clock that will be used to synchronize the data within the disc drive electronics is removed. In this way the logic 1's and 0's representing the data are taken from the disc at a time dependent on the clock generated by the disc itself. This means you can transfer a disc encoded on one drive

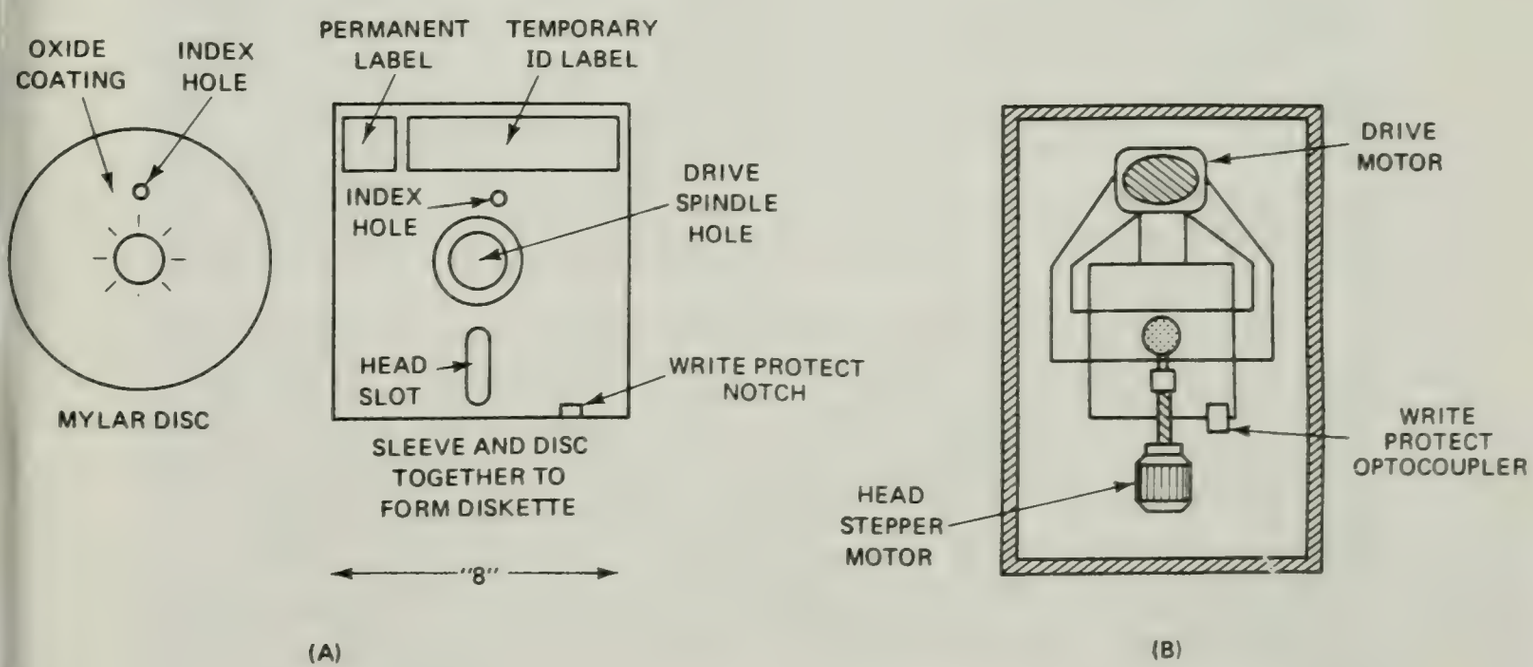


FIG. 6-8. Floppy diskette (A) and simplified disc drive mechanics at (B).

to another drive that may be running faster or slower than the original disc drive without errors.

Typical specifications for a floppy disc would be: 77 tracks per diskette; 16 hard sectors per track; 256 bytes per sector; 315,392 bytes per diskette; 3,200 bits per inch (BPI) recording density (6,400 BPI for double density); and such speed and performance specifications as 360 RPM rotational speed; 8 millisecond track-to-track access time; 8 millisecond settling time (the time it takes the head to settle once it is engaged against the diskette); 40 millisecond engage time; 10 millisecond sector read/write time; 250 millisecond average access time; and the possibility of using a dual head system, wherein one head reads and writes data from the top of the disc while the other head reads and writes data on the bottom of the disc.

The disc media itself consists of a round platter of mylar coated on one or both sides with an oxide recording surface. The read/record head is similar to those used in tape recorders (except with characteristics that allow it to record digital rather than audio). The plastic platter is inserted into a low friction envelope that protects it from external damage, dust, etc. There is a head slot provided in the envelope, cut long and narrow so the disc drive head can move across the surface of the disc and read and record data from the system. The envelope also has holes that allow the light generator/pickup assembly to shine through the index holes in the plastic platter to provide synchronizing drive pulses. The index holes on the envelope and on the drive are offset to keep the disc from being inserted in the wrong direction. As is the case with cassette tape recorders, a built-in write/protect is available on the envelope which keeps the user from being able to accidentally write data on a disc which has valuable programs that must be protected.

Two sizes of disc and disc drive are available. The standard size floppy disc is eight inches on a side, with the minifloppy using a diskette that is only 5¼ inches. On both the standard and minifloppy, the first track is labeled "track 00" and is located at the outside of the disc. The last track is located on the inside. Minifloppies generally only use thirty-five tracks total.

The sector recording format in the typical hard-sectored floppy disc will go something like that shown in the figures with the sector pulse first, then a preamble, then a sync signal, then track ID data, followed by sector ID data, followed by 256 data bytes (128 data bytes, if 32 sector holes are used), followed by checksum bytes, followed by a postamble. At the end of the postamble the next sector pulse occurs. At the beginning of each sector, thirty-two bytes of 00 are written to allow the electronic circuits in the disc controller to get themselves into synchronization after the head has moved to a new sector area. Following the preamble, one byte of FF (in hexidecimal representing eight logic 1 bits in a row) is written to serve as a synchronization word

for the data which is to follow. Next, track and sector identification information, which is dependent on the location determined by the controlling electronics and software, is included. The 256 data bytes given in this example would be any form of data that is to be stored on the diskette. As each data byte is written onto the controller, a two-byte (four-digit hexidecimal) checksum is computed by simply adding the equivalent numbers created by the data. This checksum is then stored in the checksum area and followed by the postamble which covers the remainder of the sector with logic 0's. During read-back operations, the track and sector information read from the diskette is compared with the requested track and sector to insure the proper sector is selected. In addition, as data is read from the controller, a new checksum is computed and compared with the checksum recorded on the diskette. If the two values do not agree then a checksum error has been detected. The software controlling the disc (usually called "disc driver software") would generally reread the disc several times when a checksum error occurs. Rereading would compensate for any temporary problems which might have caused the read error on the diskette. If the checksum problem is not able to be resolved, the software will flag the operator indicating to him that a checksum error has occurred and generally telling the user what track and sector are involved.

Any temporary error encountered in reading data from a disc is usually called a "soft error" and is usually corrected by simply rereading or moving the head back and forth once. If this operation doesn't correct the error, the error is decided to be a hard error which results in unrecoverable data. There are basically three types of error: Write, Read, and Seek. With the Write errors you use a Write Check Procedure where the drive reads the data again after recording during the next disc revolution. Normally, if this process shows an error you simply Write the data again. If after some predetermined number of tries (generally ten) the data is still written incorrectly, you must consider the track or sector damaged and unusable.

Read errors have already been mentioned; they too divide further into soft and hard varieties. Seek errors result when the head has not reached correct track. Seek errors can be verified by reading track and sector ID data at the beginning of the track. Whenever such an error occurs, you move the head back to track 00 and start all over again. Again after so many tries, one will generate a seek error message telling the operator that an error exists on the diskette which cannot be cleared by simply rereading data. The checksum error detection is called "CRC," or Cyclic Redundancy Checking. The controller electronics will generally provide at least six essential signals to allow communication with the disc drive.

First, Motor On. This signal also turns the motor off. You allow one second after activation when turning the motor on and deactivate the

drive after two seconds or ten revolutions whenever no further commands are issued. This saves wear and tear on the disc drive motor. Some floppy disc drives for personal computing systems allow the disc drive motor to run all the time. This means the plastic platter is turning within the protective envelope at all times. Even though the envelope is designed to be a low friction affair, there will still be wear on the oxide coating of the platter. The motor on signal merely minimizes this wear.

The second signal needed to drive a floppy disc system properly is the Direction Select signal. This signal sets the direction in which the head will move. Third, the Step Signal moves the head one track toward the disc's center or away from it. Fourth, Write Gate. When this line goes active a write operation is enabled. When it goes inactive a read occurs. If the diskette envelope write/protect hole is open, this signal will have no effect. The fifth required signal is Track 00. This indicates that the head has reached the outside track of the diskette and will move no further outward even if an additional step command is issued. Finally, the Index/Sector signal results whenever the drive detects an index hole (or a sector hole in the case of hard sectoring).

Typically the operating sequence would go something like this: the Disc Controller activates the Drive Select (usually controllers oversee more than one unit so it enables whichever drive has been selected for access). Then the controller sets the Direction Select which latches the head's direction of movement; as a result the head will move either toward the disc's center or toward the outer part of the disc. The write gate would first go inactive so that no writing would occur during head movement. The controller then pulses the step line until the head reaches the desired track. At this point the write gate may be enabled and data pulsed in on the data line or the write gate may remain disabled and data can be read out on the data line. The controller electronics will interface the head with the outside world so that the data from the source will be properly encoded as it is recorded, or decoded as it is read from the diskette.

Index

How to Debug Your Personal Computer

Jim Huffman/Robert C. Bruce

Don't kick the computer! Programs that should but won't run on your personal computer are a common source of frustration, especially if you are just beginning to write your own program or have spent hard cash for some new software. You don't have to start again at Square One or return the program. Instead, learn how to debug it yourself! This handy guide will show you how to recognize, track, and eliminate bugs in your program or system. If you have some knowledge of BASIC, you can put the techniques in this book to work right away. The step-by-step instructions are easy to follow and implement. If you're writing a program, you'll learn how to get rid of bugs as you go along and how to spot them during the shakedown period. There are also techniques for finding bugs in a program you didn't write and practical guides to help you determine where the bug is hiding and how to eliminate it or work around it. You'll find a host of valuable tips on:

- Debugging by hand calculation
- Debugging the flowchart
- Using print statements to track even the most elusive bug
- Adding program patches where nothing else will work
- Where and how each debugging tool can be used most effectively
- How to make sure your hardware is working properly

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0-8359-2924-8